Hochschule Konstanz
Fakultät Informatik

Deep Learning:: Generative Models

Dozenten: Prof. Dr. M. O. Franz, Prof. Dr. O. Dürr

Remark: Much of the material have been developed together with Beate Sick and Elvis Murina (ZHAW/Uni Zurich)

Outline and further resources

- Introduction to generative models
- Variational Autoencoders VAE*
- Generative Adversarial Networks GAN*
- Normalizing Flows

Much material taken from CS231n(2019) lecture 11
Slides: http://cs231n.stanford.edu/slides/2019/
Video: https://www.youtube.com/watch?v=5WoItGTWV54&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=14&t=0s

For VAE you might also see VAE in https://tensorchiefs.github.io/bbs/

For GANs you might also see
Jakub Langr, Vladimir Bok (GANs in Action Manning) https://www.manning.com/books/gans-in-action

State of the Art in generative models

Figure 1.1 Progress in human face generation
(Source: "The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation," by Miles Brundage et al., 2018, https://arxiv.org/abs/1802.07228.)

Credit: Jakub Langr, Vladimir Bok (GANs in Action Manning) https://www.manning.com/books/gans-in-action

State of the art (2019) in generative models

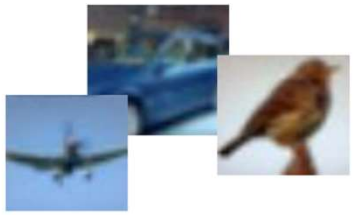
https://thispersondoesnotexist.com/

These persons are not real. The pictures **1024x1024x3** are sampled from a GAN.

Do you spot some artefacts?

Definition: Generative Model [cs231n]

Given training data, generate new samples from same distribution.



Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Several flavors:

- Explicit density estimation: explicitly define and learn $p_{model}(x)$
- Implicit density estimation: learn model that can sample from $p_{model}(x)$ w/o explicitly having a density

Why Generative Models?

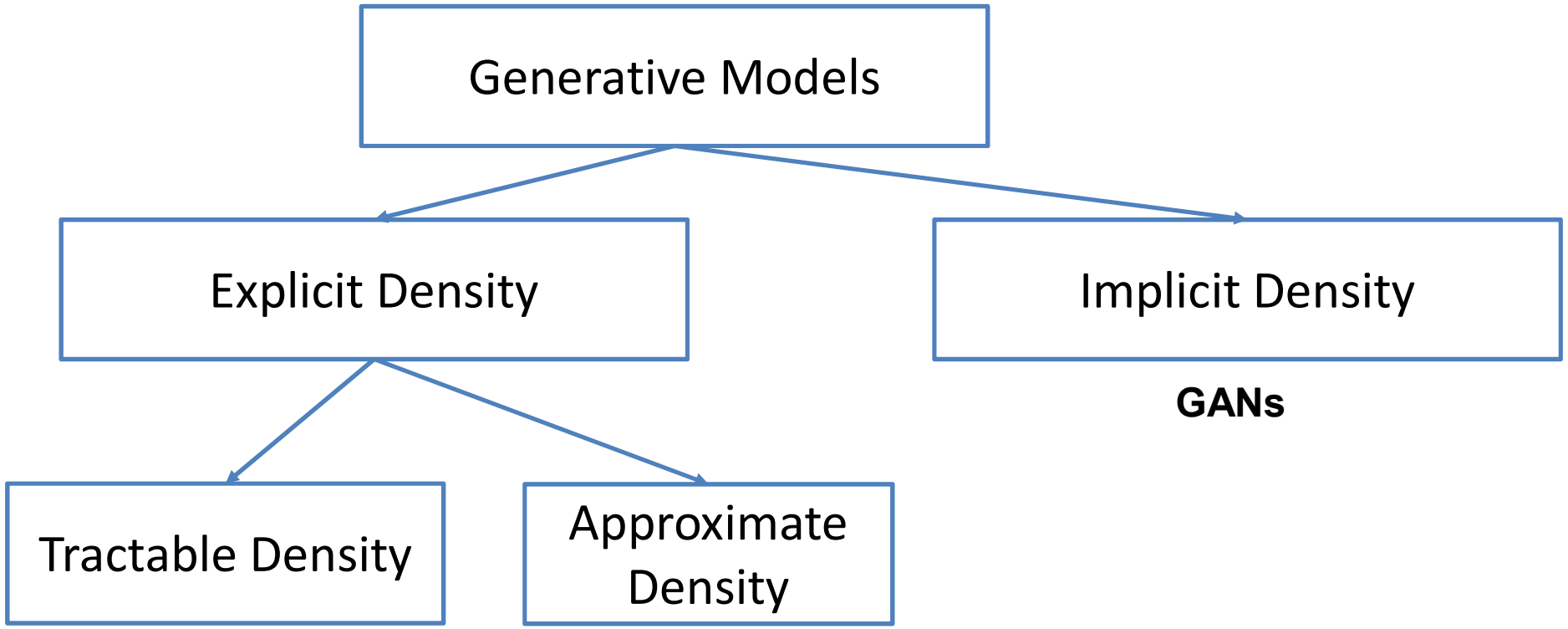
- Generation of new data
  - For fun create persons that does not exists
  - Additional training data
  - Private Data (anonymization)
  - Image and Audio synthesis Wavenet / PixelCNN

- Outlier detection learn $p_{ok}(x)$
  - Is image/vibration/… *x* from ok distribution?
    - Best with explicit models

- Other
  - Learn Latent representation (e.g. make people smile)
  - Semi-supervised learning

Image from Priyank Jaini talk

Taxonomy of Generative Models



Normalizing Flows

- PixelCNN
- Wavenet
- Glow
- Nice RealNVP

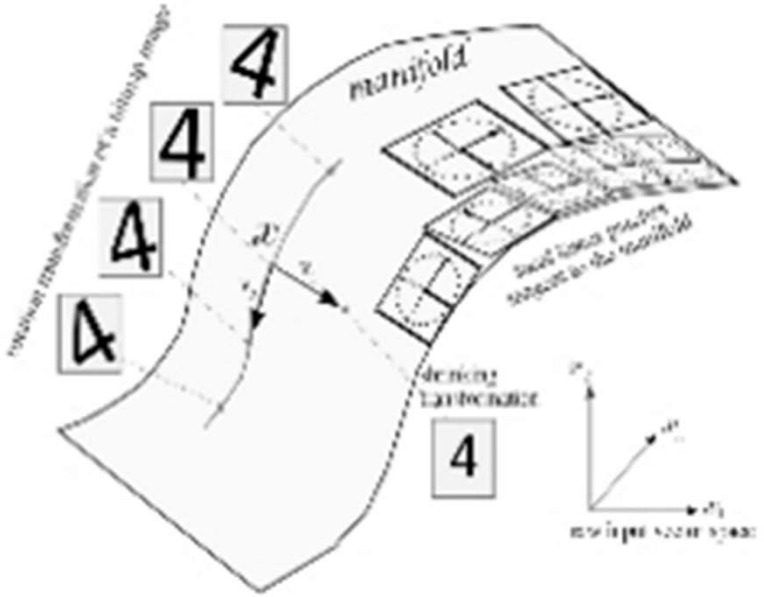
Variational Autoencoder

Figure simplified from CS231n

Variational Autoencoder

Backgound: Manifold hypothesis

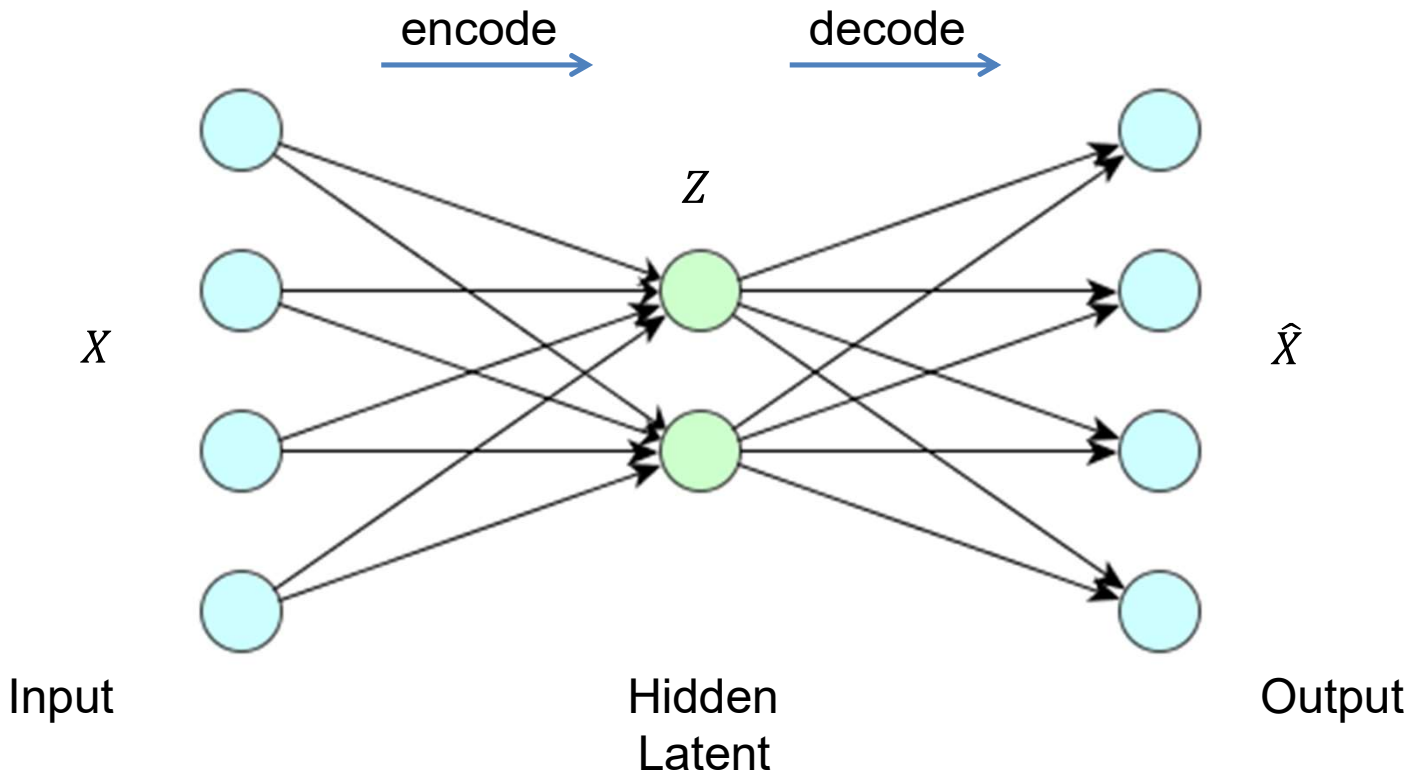
- X high dimensional vector
- Data is concentrated around a low dimensional manifold



- Hope finding a representation Z of that manifold.

credit: http://www.deeplearningbook.org/

Auto Encoders ('linear classical')



You already now an autoencoder?

- PCA
- In PCA the training is done so that reconstruction error $\|X - \hat{X}\|^2$ is minimized
- Unsupervised methods (no labels required)

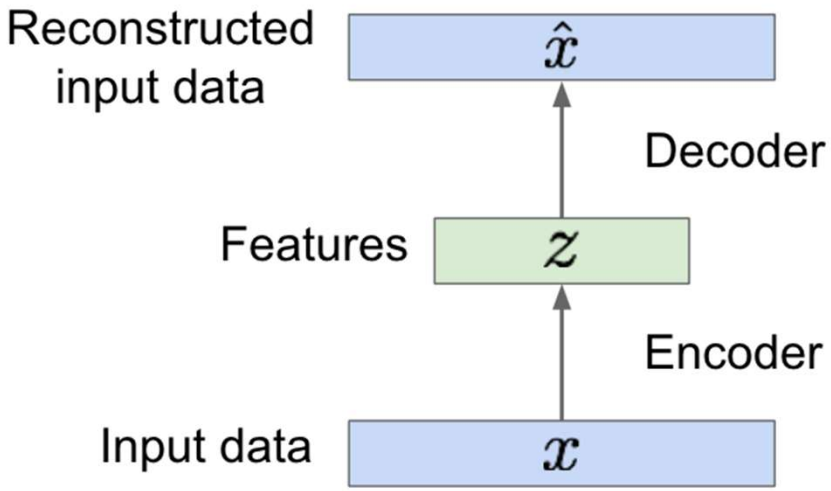
Advanced autoencoders with CNNs

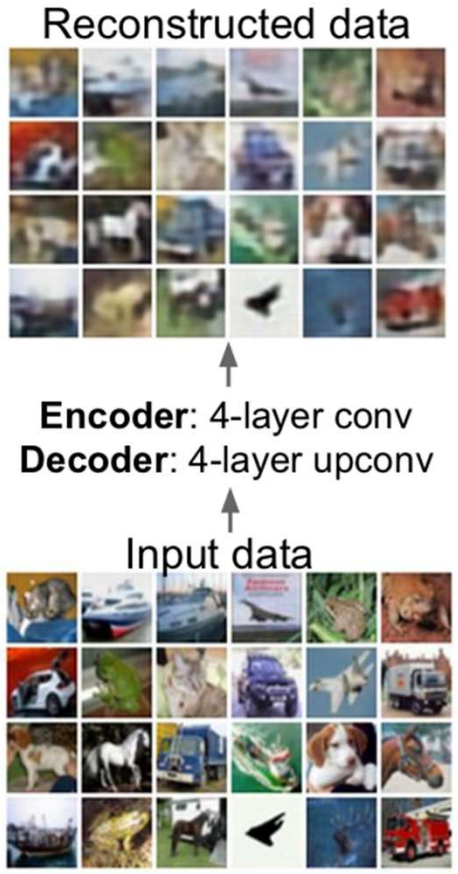
Some background first: Autoencoders

How to learn this feature representation?

Train such that features can be used to reconstruct original data

"Autoencoding" - encoding itself





Use non-linearities and convolutions for images

Figure from CS231n

Application of Autoencoders

- Semi-supervised learning (typical real-live scenario)
  - Many unlabeled data (like images)
  - Few labels (labeling is expensive)
- Idea
  - The latent representation *z* of an image is more compact than the image itself
  - Learning the encoder from x → z does not require labels
  - Two step procedure
    1. Learn the encoder from x →z
    2. Lean the classificatory on z instead of x

Application: Semi-Supervised Learning

Step 1: Training the Encoder and Decoder Network (no labels requires)

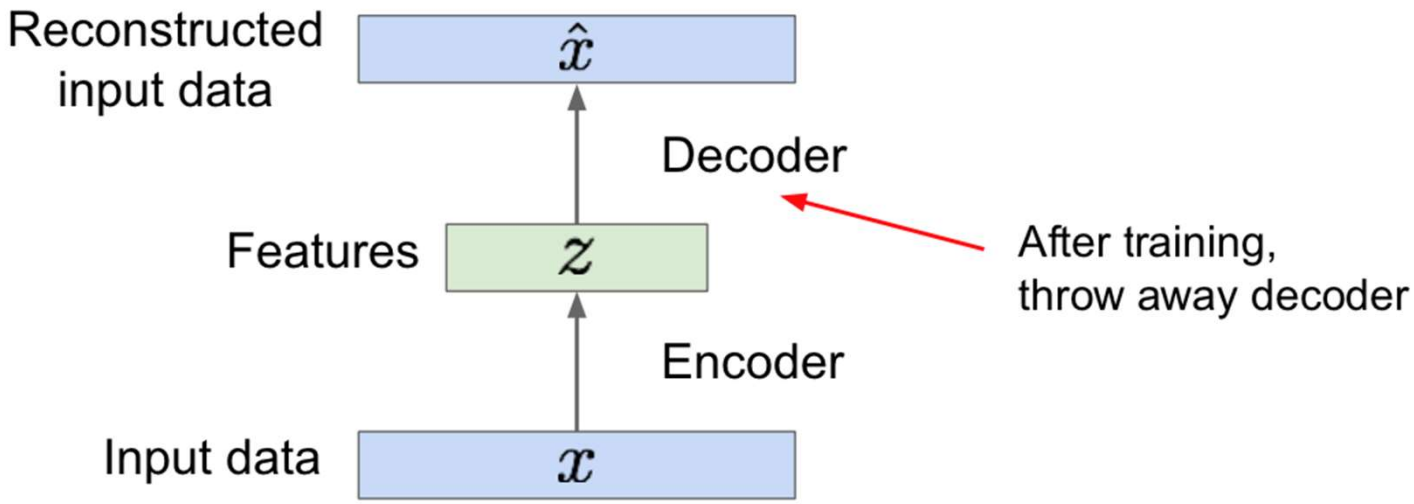
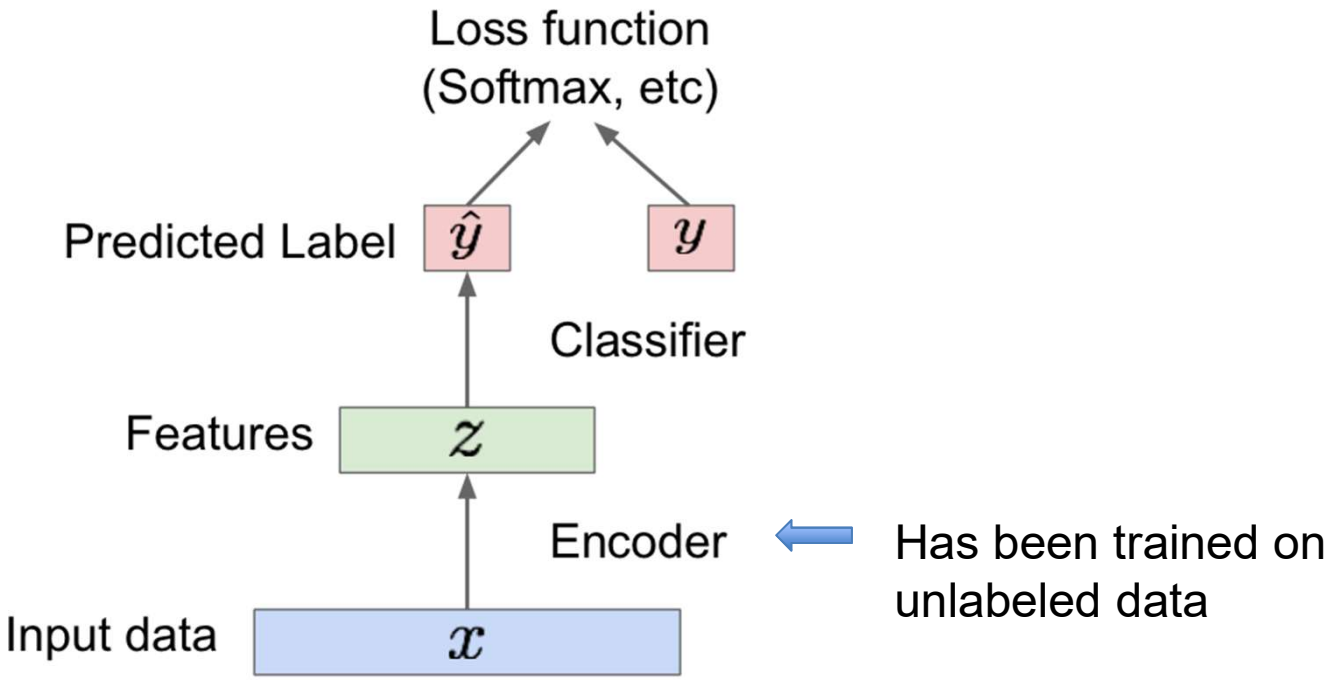


Figure from CS231n

Application: Semi-Supervised Learning

Step 2: Train Classifier on compact latent representation z of x for which we have labels y



Encoder is fix or can also be fine tuned.

Figure from CS231n

Variational Autoencoders ("history")

Simultaneously discovered by

- Kingma and Welling. "*Auto-Encoding Variational Bayes*, *International Conference on Learning Representations." ICLR,* 2014. arXiv:1312.6114 [stat.ML] (20 December 2013, Amsterdam University) Talk

- Rezende, Mohamed and Wierstra. "*Stochastic back-propagation and variational inference in deep latent Gaussian models." ICML, 2014arXiv:1401.4082 [stat.ML] (16 January 2014, Google DeepMind)*

Alternative approach (for binary distributions)

- *Gregor, Danihelka et all. "Deep autoregressive networks." ICML 2014*
  - Has a more information theoretic ansatz (codings length)
  - Lecture given at *Nando de Freitas ML Course (University of Oxford) (a bit hand waving argument but with nice examples)*

- We focus on the approach as in Kingma, Welling

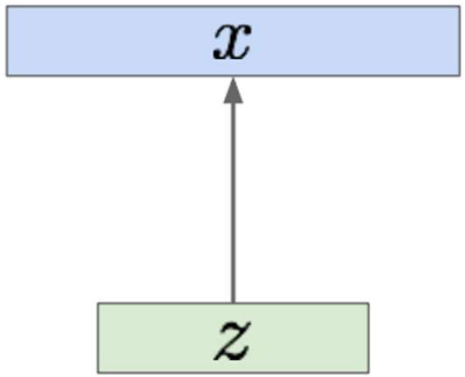
Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

Assume training data $\{x^{(i)}\}_{i=1}^{N}$ is generated from underlying unobserved (latent) representation **z**

Sample from true conditional $p_{\theta^*}(x \mid z^{(i)})$

Sample from true prior $p_{\theta^*}(z)$



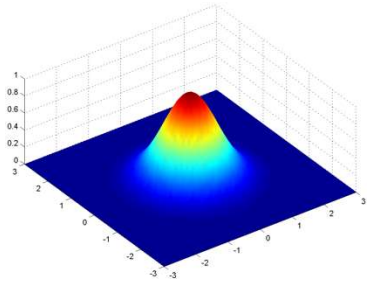
Sampling from $p(x)$ is a two step procedure

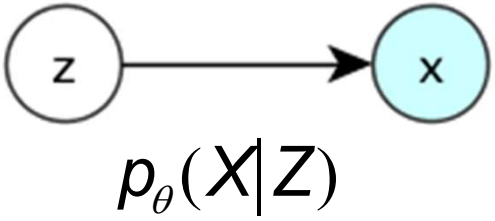
1. Sample from latent z $z^{(i)} \sim p_\theta(z)$
2. Sample from $z^{(i)} \sim p_\theta(x|z^{(i)})$

Slide from CS231n

Modeling the decoder

- Two parts
  - $p_\theta(z)$ use a simple low dimensional distribution (e.g. univariate Gaussian)

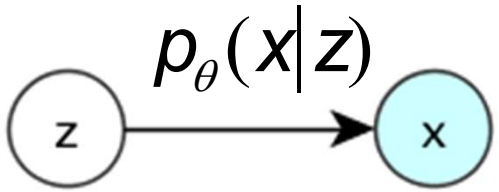




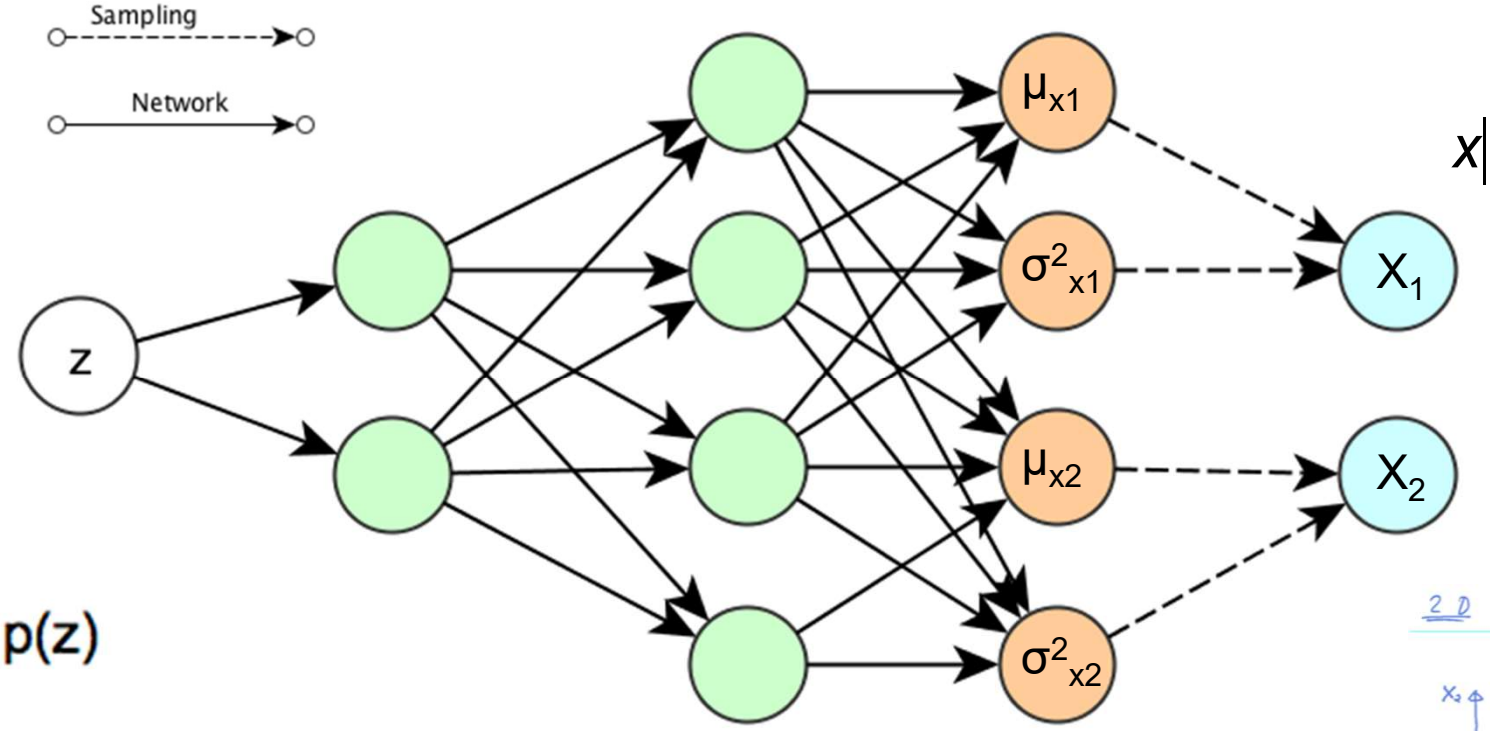
One Example

  - $p_\theta(z|x)$ goes from 2-D to images needs to be complex
    - As in last section, we use a network to control the parameters of a distribution

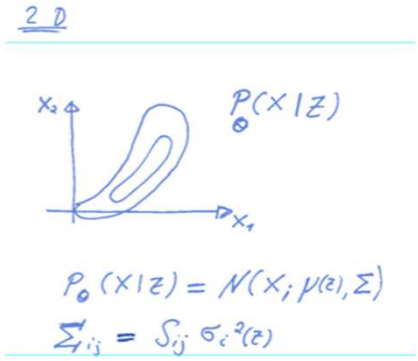
The model for the decoder network



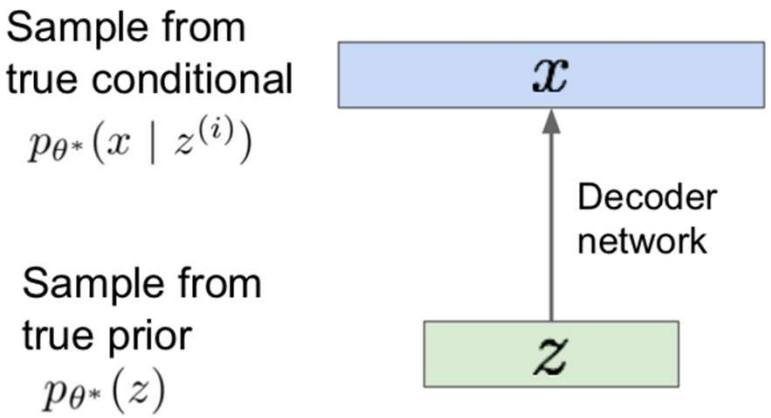
- For illustration z one dimensional x 2D
- Want a complex model of distribution of x given z
- Idea: **NN** + Gaussian (or Bernoulli) here with diagonal covariance Σ



$$x|z \sim N(\mu_x, \sigma_x^2)$$



Training (directly)



We want to estimate the true parameters θ^* of this generative model.

How to train the model?

Remember strategy for training generative models from FVBNs. Learn model parameters to maximize likelihood of training data

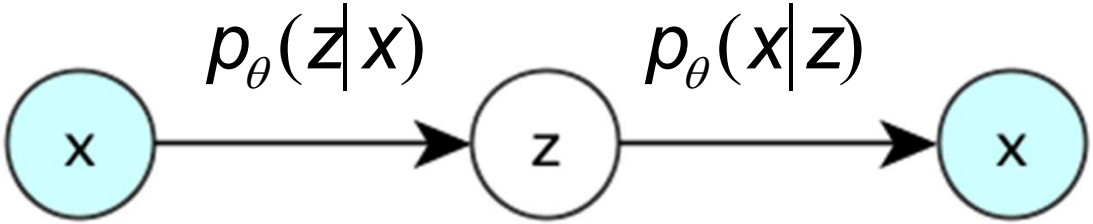
$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

What is the problem?

- Integral is intractable
- $p_\theta(x|z)$ and $p_\theta(z)$ OK

Figure from CS231n

Training as an autoencoder



Training use maximum likelihood of p(x) given the training data

Problem (decoder)

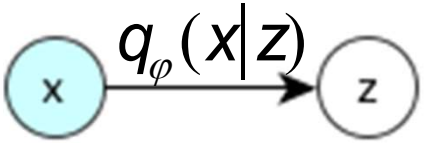
$$p_\theta(z|x)$$

Cannot be calculated:

Solution:

- MCMC (too costly)
- Approximate $\mathrm{p}_\theta(z|x)$ with $q_\varphi(z|x)$

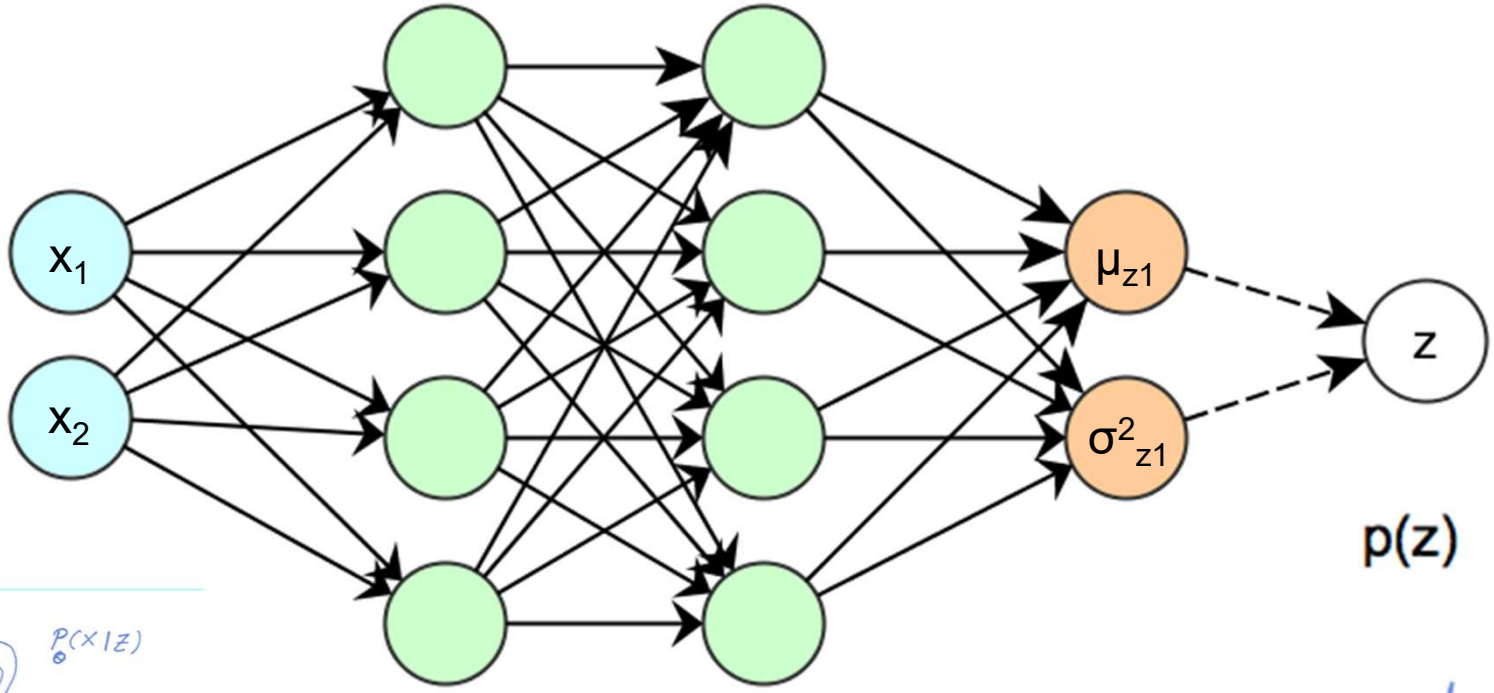
The model for the encoder network

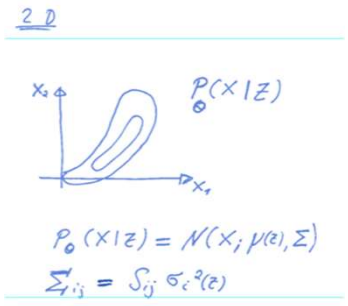


- A feed forward NN + Gaussian

$$q_\phi(z \mid x) = \mathcal{N}(z; \mu_z(x), \sigma_z(x))$$

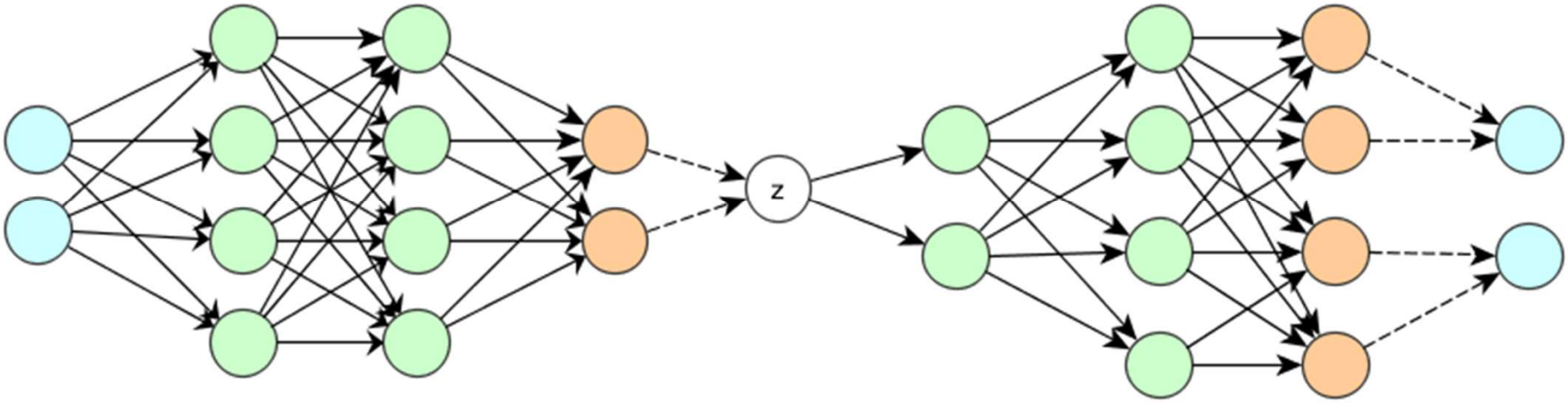
Just a Gaussian, with diagonal covariance.





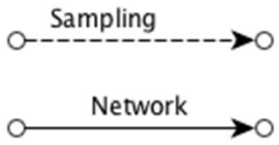
The complete auto-encoder

$q_\varphi(x|z)$ $p_\theta(x|z)$



Learning the parameters φ and θ via backpropagation

Determining the loss function



Some math recap?

- Blackboard
  - Expectation as Integral
  - KL-Divergence

Training with maximum likelihood

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\log p_\theta(x^{(i)}) = \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z)$$

Trick!

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Bayes' Rule)}$$

$p(x|z) = p(z|x)p(x)/p(z)$

We want to maximize the data likelihood

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \quad \text{(Multiply by constant)}$$

$$= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Logarithms)}$$

$$= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z)) + D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))$$

Decoder network gives $p_\theta(x|z)$, can compute estimate of this term through sampling. (Sampling differentiable through reparam. trick, see paper.)

This KL term (between Gaussians for encoder and z prior) has nice closed-form solution!

$p_\theta(z|x)$ intractable (saw earlier), can't compute this KL term :(But we know KL divergence always >= 0.

$$E_{z\sim q(z|x)}[\log(\frac{q(z|x)}{p(z)})] = \int q(z|x)\log\left(\frac{q(z|x)}{p(z)}\right)dz = D_{KL}(q(z|x)||p(z))$$

Figure from CS231n

Training with maximum likelihood

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

We want to maximize the data likelihood

$$\log p_\theta(x^{(i)}) = \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z)$$

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Bayes' Rule)}$$

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \quad \text{(Multiply by constant)}$$

$$= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Logarithms)}$$

$$= \underbrace{\mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)} + \underbrace{D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))}_{\geq 0}$$

Tractable lower bound which we can take gradient of and optimize! ($p_\theta(x|z)$ differentiable, KL term differentiable)

$L(x^{(i)}, \theta, \phi)$ is lower bound of $\log(p_\theta(x^{(i)})$ called evidence. Hence $L(x^{(i)}, \theta, \phi)$ is also referred to as ELBO (Evidence Lower Bound)

Figure from CS231n

Training with maximum likelihood

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\log p_\theta(x^{(i)}) = \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z)$$

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Bayes' Rule)}$$

Reconstruct the input data

Make approximate posterior distribution close to prior

$$= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \quad \text{(Multiply by constant)}$$

$$= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Logarithms)}$$

$$= \underbrace{\mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z))}_{\mathcal{L}(x^{(i)},\theta,\phi)} + \underbrace{D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))}_{>0}$$

$$\log p_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)},\theta,\phi)$$

Variational lower bound ("ELBO")

$$\theta^*, \phi^* = \arg\max_{\theta,\phi} \sum_{i=1}^{N} \mathcal{L}(x^{(i)},\theta,\phi)$$

Training: Maximize lower bound

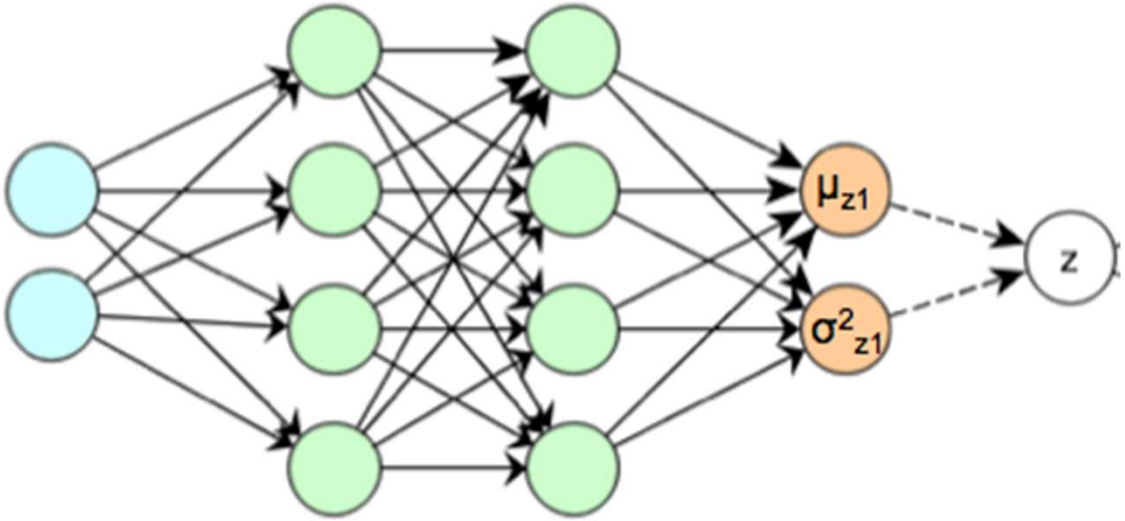
Figure from CS231n

Calculation the regularization $-D_{\mathrm{KL}}\left(q(z|x^{(i)})\|p(z)\right)$

Use N(0,1) as prior for p(z)

$q(z|x^{(i)})$ is Gaussian with parameters $(\mu^{(i)},\sigma^{(i)})$ determined by NN

$$-D_{\mathrm{KL}}\left(q(z|x^{(i)})\|p(z)\right) = \frac{1}{2}\sum_{j=1}^{J}\left(1+\log(\sigma_{z_j}^{(i)^2}) - \mu_{z_j}^{(i)^2} - \sigma_{z_j}^{(i)^2}\right)$$



Sampling to calculate $\mathbb{E}_{q(z|x^{(i)})}\left(\log\left(p(x^{(i)}|z)\right)\right)$

Approximating $\mathbb{E}_{q(z|x^{(i)})}$ with sampling form the distribution $q(z|x^{(i)})$

With $z^{(i,l)}$ $l = 1, 2, \ldots L$ sampled from $z^{(i,l)} \sim q(z|x^{(i)})$

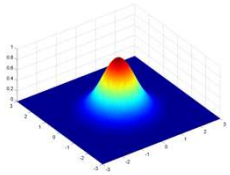
$$L^{v} = -D_{KL}\left(q(z|x^{(i)})||p(z)\right) + \mathbb{E}_{q(z|x^{(i)})}\left(\log\left(p(x^{(i)}|z)\right)\right)$$

$$L^{v} \approx -D_{KL}\left(q(z|x^{(i)})||p(z)\right) + \frac{1}{L}\sum_{i=1}^{L}\log\left(p(x^{(i)}|z^{(i,l)})\right)$$

Example $x^{(i)}$

$q_\phi(z|x^{(i)})$

z



$\log(p_\theta(x^{(i)}|z^{(i,1)}))$ where $z^{(i,1)} \sim N(\mu_z^{(i)}, \sigma_z^{2(i)})$

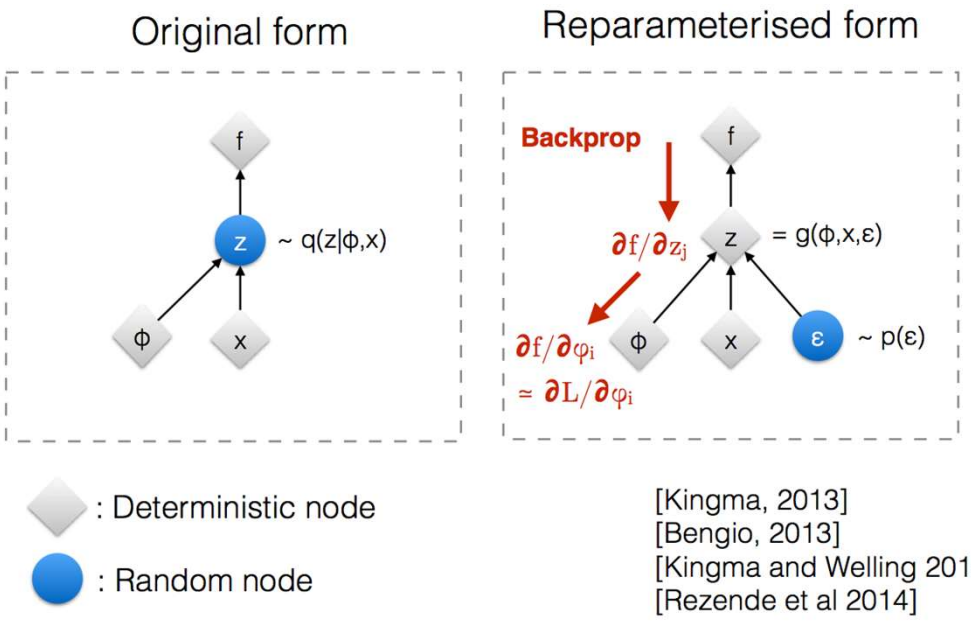
...

$\log(p_\theta(x^{(i)}|z^{(i,L)}))$ where $z^{(i,L)} \sim N(\mu_z^{(i)}, \sigma_z^{2(i)})$

L is often very small (often just L=1)

One last trick

Backpropagation not possible through random sampling!



[Kingma, 2013]
[Bengio, 2013]
[Kingma and Welling 2014]
[Rezende et al 2014]

Sampling (reparametrization trick)

$$z^{(i,l)} \sim N(\mu^{(i)}, \sigma^{2(i)})$$

$$z^{(i,l)} = \mu^{(i)} + \sigma^{(i)} \odot \varepsilon_i \quad \varepsilon_i \sim N(0,1)$$

Cannot back propagate through a random drawn number

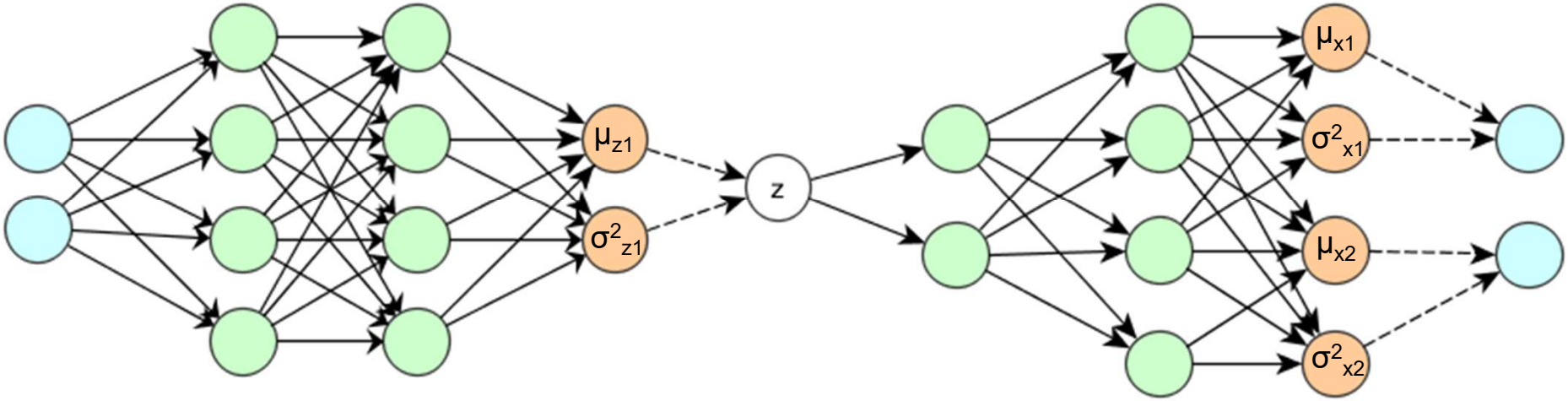
z has the same distribution, but now one can back propagate.

Writing z in this form, results in a deterministic part and noise.

Image from: NIPS Workshop 2015 (Kingma & Welling)

Putting it all together

Prior p(z) ~ N(0,1) and p, q Gaussian, extension to dim(z) > 1 trivial



Cost: Regularisation

$$-D_{\mathrm{KL}}\left(q(z|x^{(i)})\|p(z)\right) = \frac{1}{2}\sum_{j=1}^{J}\left(1+\log(\sigma_{z_j}^{(i)^2}) - \mu_{z_j}^{(i)^2} - \sigma_{z_j}^{(i)^2}\right)$$

Cost: Reproduction

$$-\log\left(p(x^{(i)}|z^{(i)})\right) = \sum_{j=1}^{D}\frac{1}{2}\log(\sigma_{x_j}^2) + \frac{(x_j^{(i)} - \mu_{x_j})^2}{2\sigma_{x_j}^2}$$

We use mini batch gradient decent to optimize the cost function over all $x^{(i)}$ in the mini batch

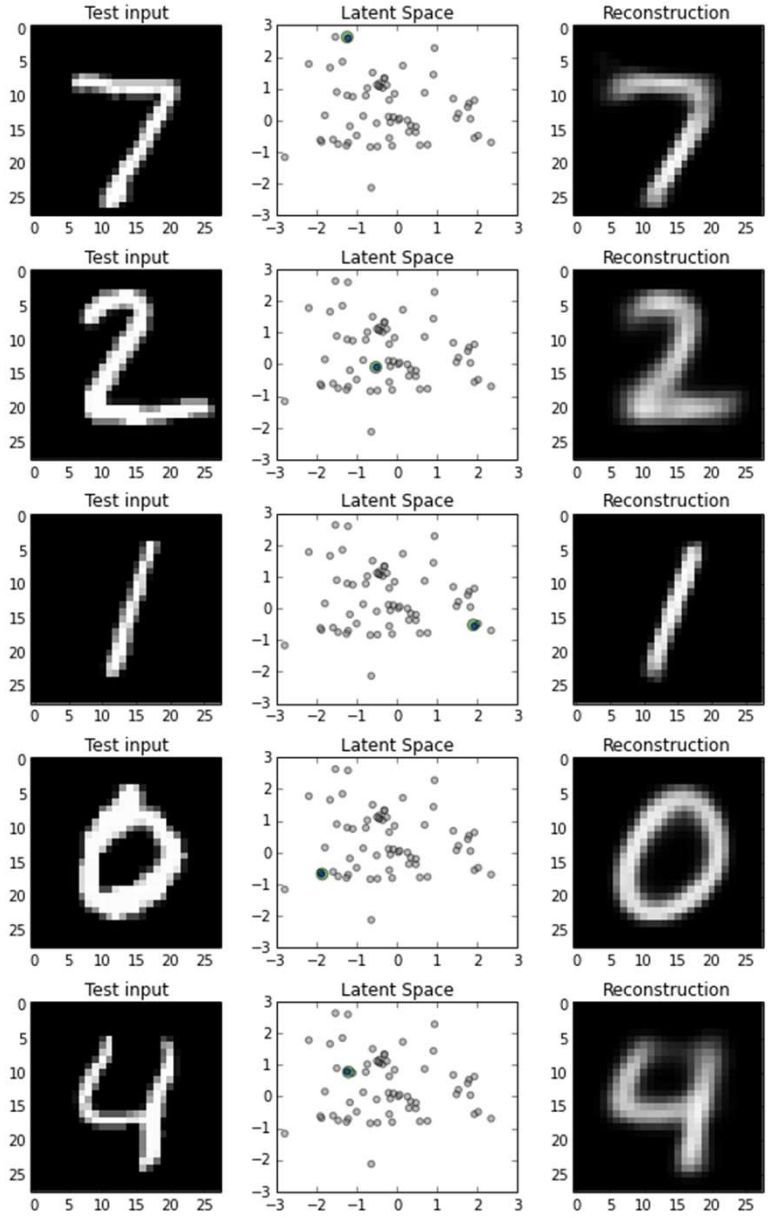
Least Square for constant variance

Results Testinput (MNIST)

Fully connected network

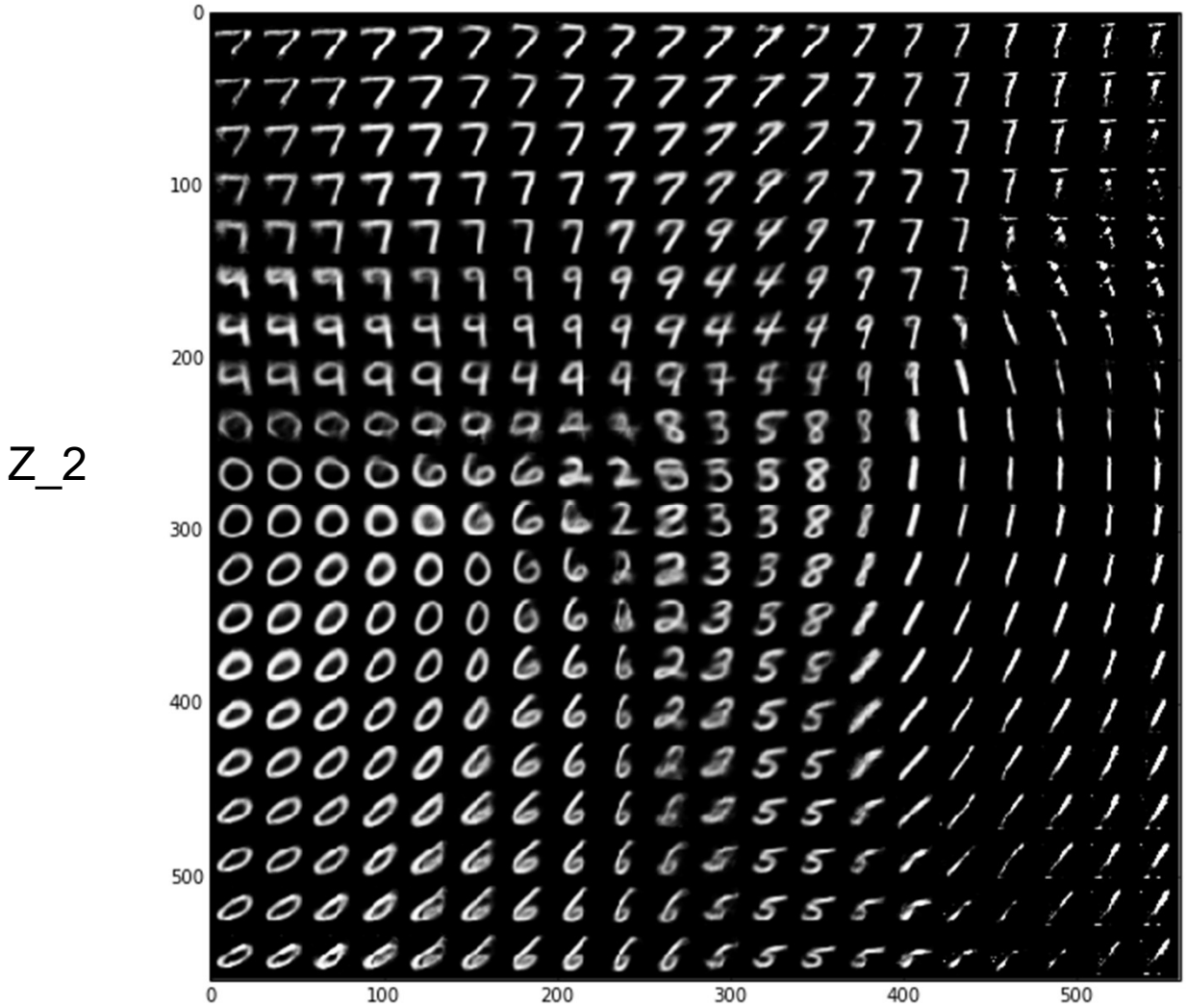
2 hidden layer (500, 501)

2 Dimensional latent space z



For a TensorFlow implementation see: https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/vae/vae_demo.ipynb

Results Moving in latent space

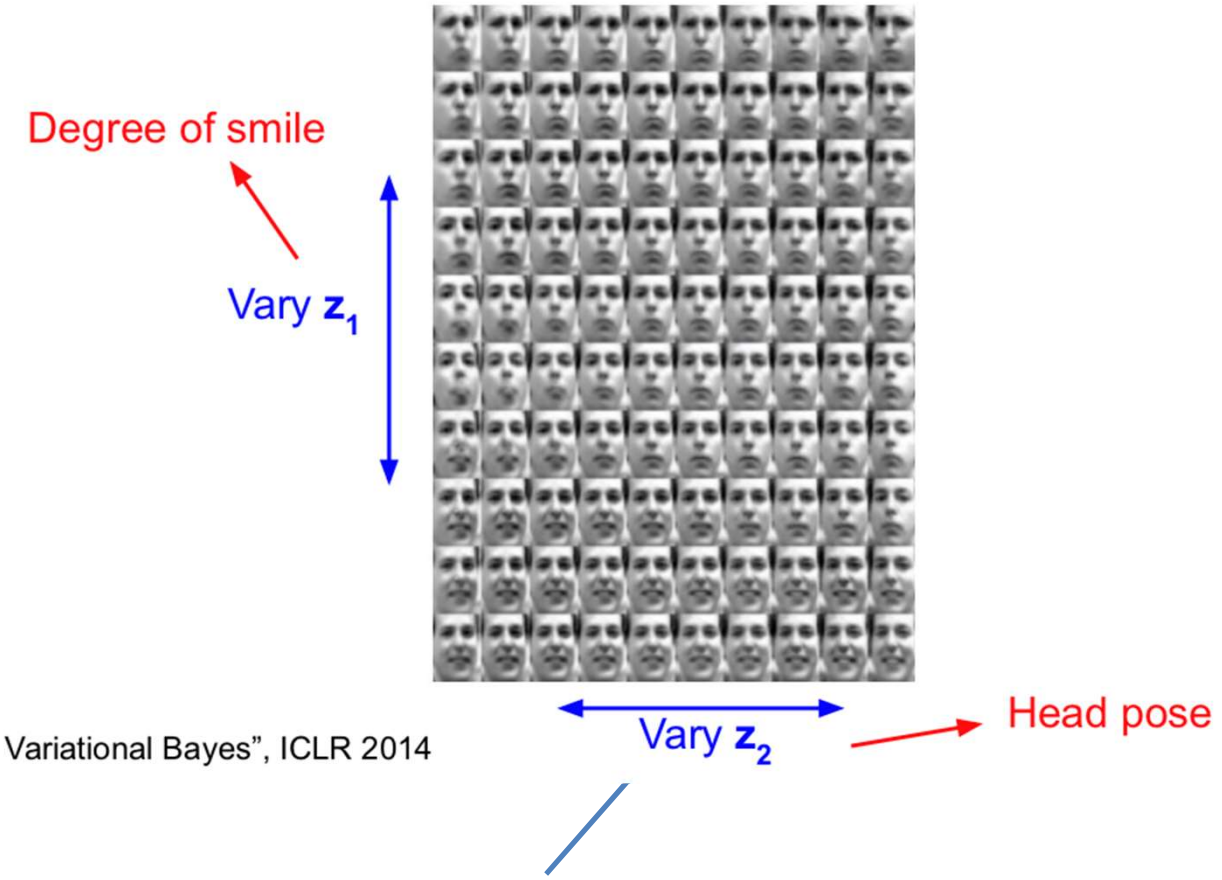


For a TensorFlow implementation see: https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/vae/vae_demo.ipynb

Generating Data (trained on faces)

Diagonal prior on **z**
=> independent
latent variables

Different
dimensions of **z**
encode
interpretable factors
of variation



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Instead of sampling,
we move in two-dimensional z-space

Figure from CS231n

State of the art VAQ (2019)

- VQ-VAE-2 https://arxiv.org/abs/1906.00446

VQ-VAE (Proposed)

BigGAN deep

Pros and Cons

Probabilistic spin to traditional autoencoders => allows generating data
Defines an intractable density => derive and optimize a (variational) lower bound

Pros:

- Principled approach to generative models
- Allows inference of q(z|x), can be useful feature representation for other tasks

Cons:

- Maximizes lower bound of likelihood: okay, but not as good evaluation as PixelRNN/PixelCNN
- Samples blurrier and lower quality compared to state-of-the-art (GANs)

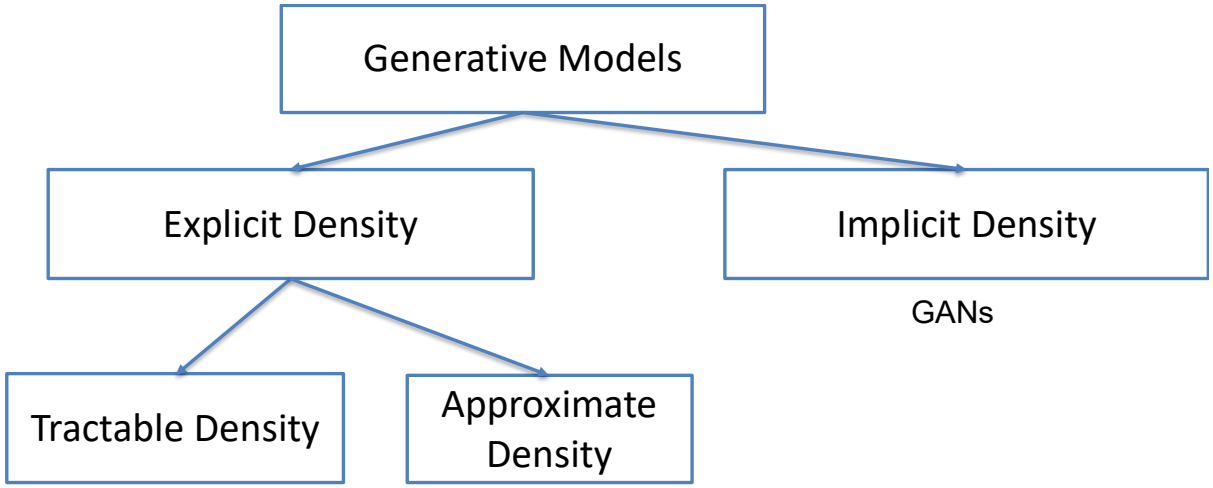
Active areas of research:

- More flexible approximations, e.g. richer approximate posterior instead of diagonal Gaussian, e.g., Gaussian Mixture Models (GMMs)
- Incorporating structure in latent variables, e.g., Categorical Distributions

From CS231n

GAN

Idea of GANs



GANs: don't work with any explicit density function!
Instead, take game-theoretic approach: learn to generate from training distribution through 2-player game

From CS231n

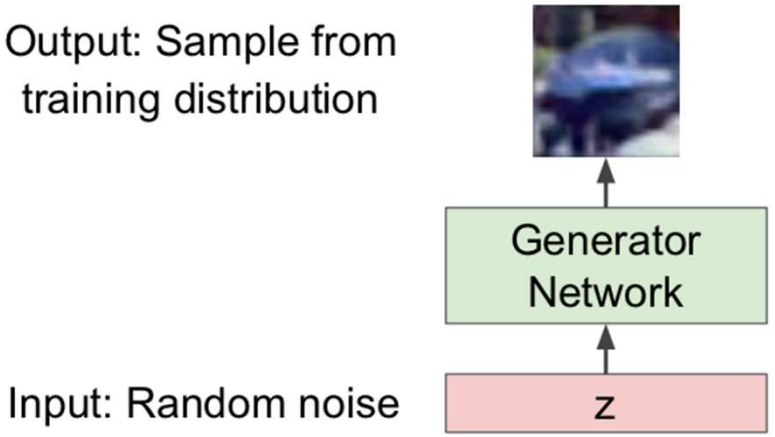
Sampling from noise

Problem: Want to sample from complex, high-dimensional training distribution. No direct way to do this!

Solution: Sample from a simple distribution, e.g. random noise. Learn transformation to training distribution.

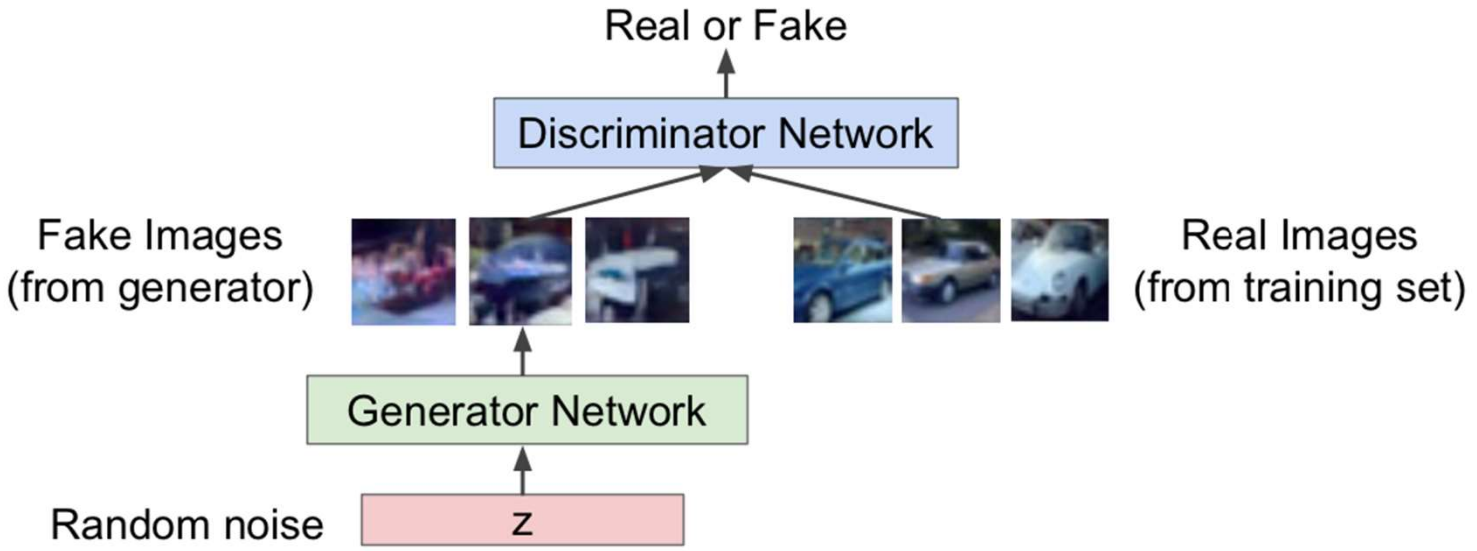
Q: What can we use to represent this complex transformation?

A: A neural network!

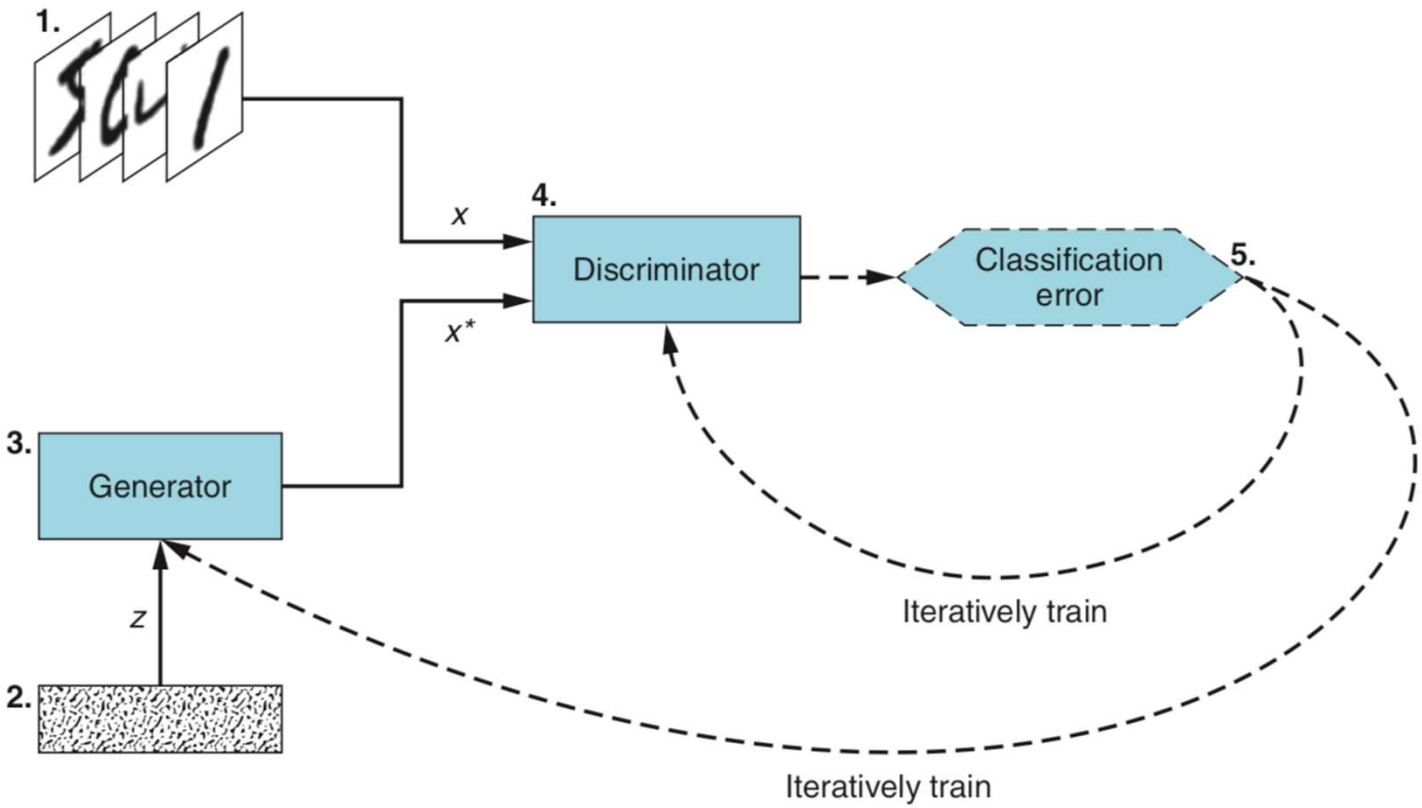


Training GANs: Two-player game

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images



Overview



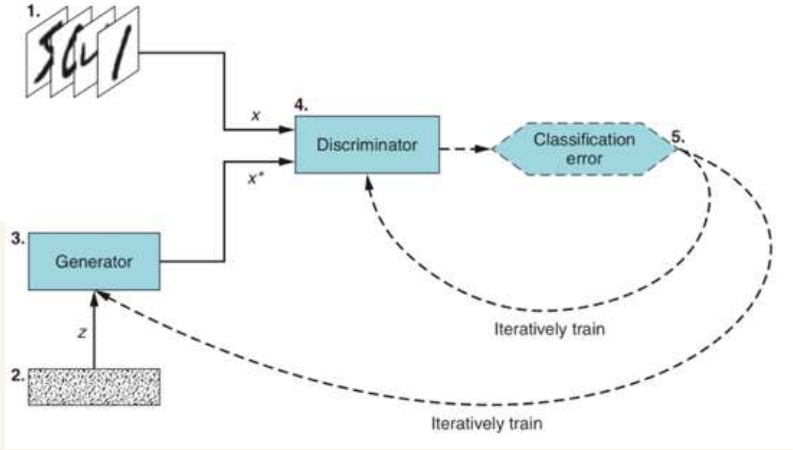
Ideal coding of Discriminator

- $D(x) = 1$ if real
- $D(x^*) = 0$ if fake

Discriminator reports likelihood [0,1] that image is real.

Credit: Jakub Langr, Vladimir Bok (GANs in Action Manning) https://www.manning.com/books/gans-in-action

Training



GAN training algorithm

For each training iteration ***do***

1. Train the Discriminator:
   a. Take a random mini-batch of real examples: x.
   b. Take a mini-batch of random noise vectors z and generate a mini-batch of fake examples: $G(z) = x^*$.
   c. Compute the classification losses for $D(x)$ and $D(x^*)$, and backpropagate the total error to update $\theta^{(D)}$ to *minimize* the classification loss.
2. Train the Generator:
   a. Take a mini-batch of random noise vectors z and generate a mini-batch of fake examples: $G(z) = x^*$.
   b. Compute the classification loss for $D(x^*)$, and backpropagate the loss to update $\theta^{(G)}$ to *maximize* the classification loss.

End for

Credit: Jakub Langr, Vladimir Bok (GANs in Action Manning) https://www.manning.com/books/gans-in-action

Cost function of the discriminator

- Using Log Likelihood in binary classification*
- Looking at the discriminator $D_{\theta_d}(x)$ a network parametrized by θ_d
- Take a single example (image)
  - If it's real x
    - $\log(D_{\theta_d}(x))$ should be maximal
  - If it's not real x*=G(z) coming from generator
    - $\log(1 - D_{\theta_d}(x^*))$ should be maximal

- The Discriminator maximizes θ_d in

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

- The Generator wants to make this expression small

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

*The GAN community maximizes Log Like instead of the usual minimization of neg. Log Like.

Training GANs: Objective

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

Discriminator outputs likelihood in (0,1) of real image

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Training Gans: procedure

Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

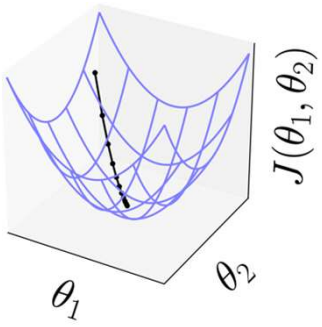
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Gradient descent** on generator

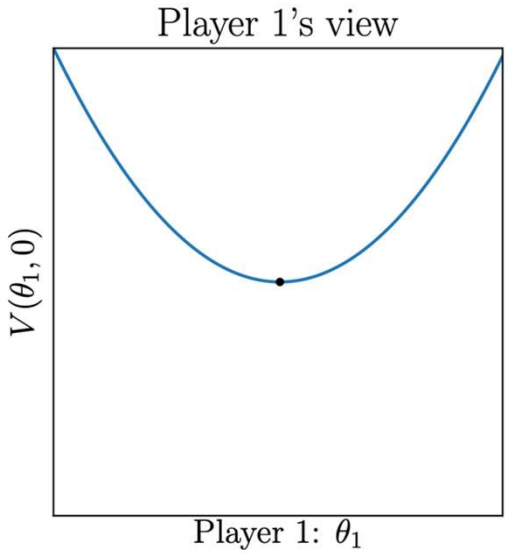
$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

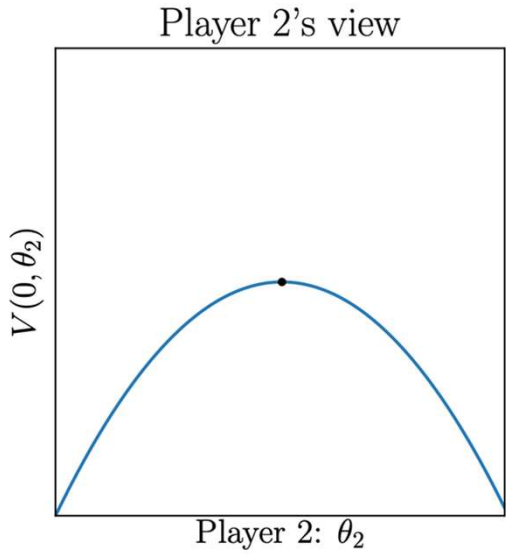
Nash equilibrium

- Most traditional ML algorithm



- Zero sum game, adversarial machine learning





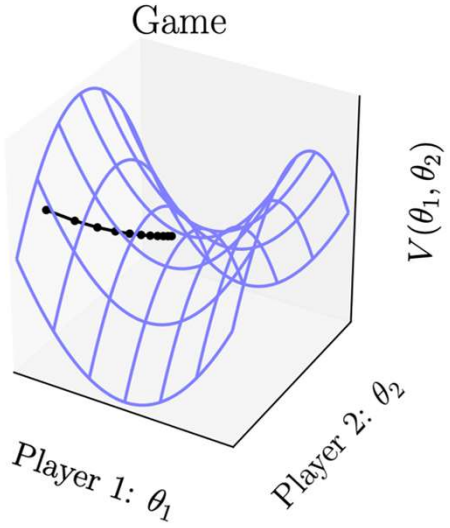


Image credit: http://www.iangoodfellow.com/slides/2019-05-07.pdf

Example Code

- Let's look at a simple implementation, to create MNIST
- Taken from chapter 3 of
  - https://www.manning.com/books/gans-in-action
- Code can be found at
  - https://github.com/GANs-in-Action/gans-in-action/blob/master/chapter-3/Chapter_3_GAN.ipynb

The generator

A network going from a `z_dim`=100 dimensional noise vector to an `img_shape=(28,28,1)` image

Listing 3.3 Generator

```
def build_generator(img_shape, z_dim):
    model = Sequential()

    model.add(Dense(128, input_dim=z_dim))

    model.add(LeakyReLU(alpha=0.01))

    model.add(Dense(28 * 28 * 1, activation='tanh'))

    model.add(Reshape(img_shape))

    return model
```

Fully connected layer

Leaky ReLU activation

Output layer with tanh activation

Reshapes the Generator output to image dimensions

All pixel values in range [-1,1] as tanh

The discriminator

A network going from `img_shape=(28,28,1)` to 1-dimensional output

Listing 3.4 Discriminator

```
def build_discriminator(img_shape):

    model = Sequential()

    model.add(Flatten(input_shape=img_shape))

    model.add(Dense(128))

    model.add(LeakyReLU(alpha=0.01))

    model.add(Dense(1, activation='sigmoid'))

    return model
```

Flattens the input image

Fully connected layer

Leaky ReLU activation

Output layer with sigmoid activation

The complete network

Listing 3.5 Building and compiling the GAN

```
def build_gan(generator, discriminator):

    model = Sequential()

    model.add(generator)
    model.add(discriminator)

    return model


discriminator = build_discriminator(img_shape)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(),
                      metrics=['accuracy'])

generator = build_generator(img_shape, z_dim)

discriminator.trainable = False

gan = build_gan(generator, discriminator)
gan.compile(loss='binary_crossentropy', optimizer=Adam())
```

Combined Generator + Discriminator model

Builds and compiles the Discriminator

Builds the Generator

Keeps Discriminator's parameters constant for Generator training

Builds and compiles GAN model with fixed Discriminator to train the Generator

Discriminator has standard loss

gan is $D(G(z))$

Listing 3.6 GAN training loop

```
losses = []
accuracies = []
iteration_checkpoints = []


def train(iterations, batch_size, sample_interval):

    (X_train, _), (_, _) = mnist.load_data()

    X_train = X_train / 127.5 - 1.0
    X_train = np.expand_dims(X_train, axis=3)

    real = np.ones((batch_size, 1))

    fake = np.zeros((batch_size, 1))

    for iteration in range(iterations):


        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]

        z = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(z)

        d_loss_real = discriminator.train_on_batch(imgs, real)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
        d_loss, accuracy = 0.5 * np.add(d_loss_real, d_loss_fake)



        z = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(z)

        g_loss = gan.train_on_batch(z, real)

        if (iteration + 1) % sample_interval == 0:

            losses.append((d_loss, g_loss))
            accuracies.append(100.0 * accuracy)
            iteration_checkpoints.append(iteration + 1)
```

Loads the MNIST dataset

Rescales [0, 255] grayscal pixel values to [–1, 1]

Labels for real images: all 1s

Labels for fake images: all 0s

Gets a random batch of real images

Generates a batch of fake images

Trains the Discriminator

Return loss and acc

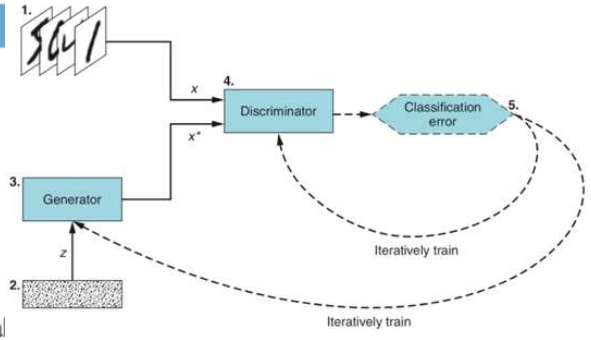
Generates a batch of fake images

The discriminator is fixed

Trains the Generator

Wrong labels, same as maximizing

Saves losses and accuracies so they can be plotted after training

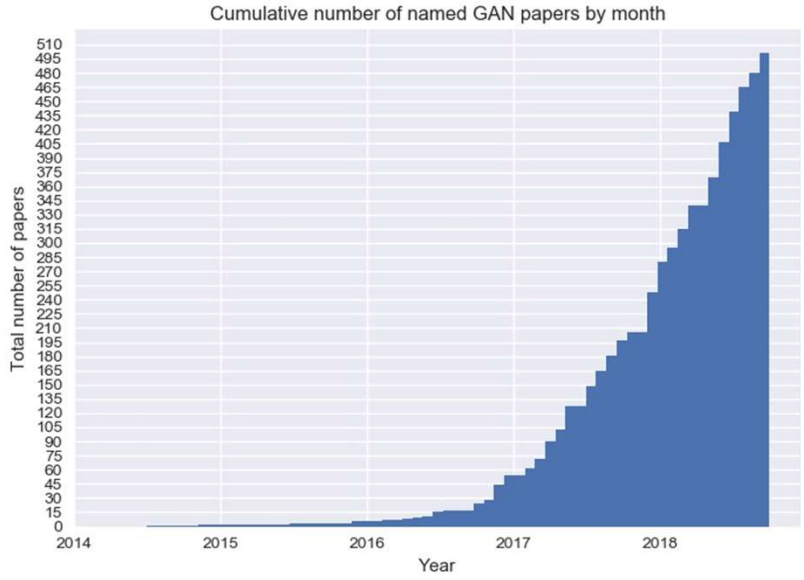


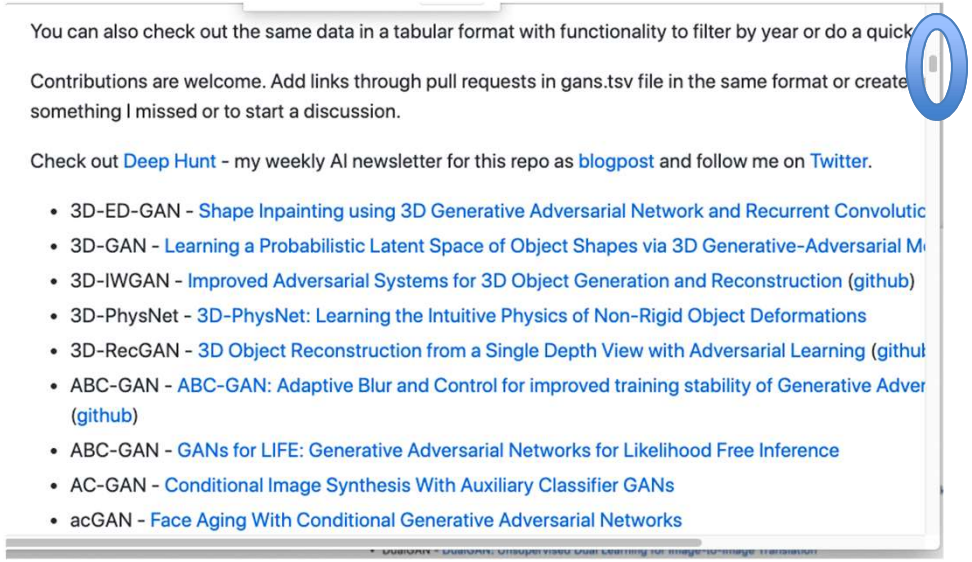
Results after 20000 iterations

Room for improvement

Enhancements / Variants

- Since ~2017 extreme number of papers on GANS
- https://github.com/hindupuravinash/the-gan-zoo





Enhancements / Variants

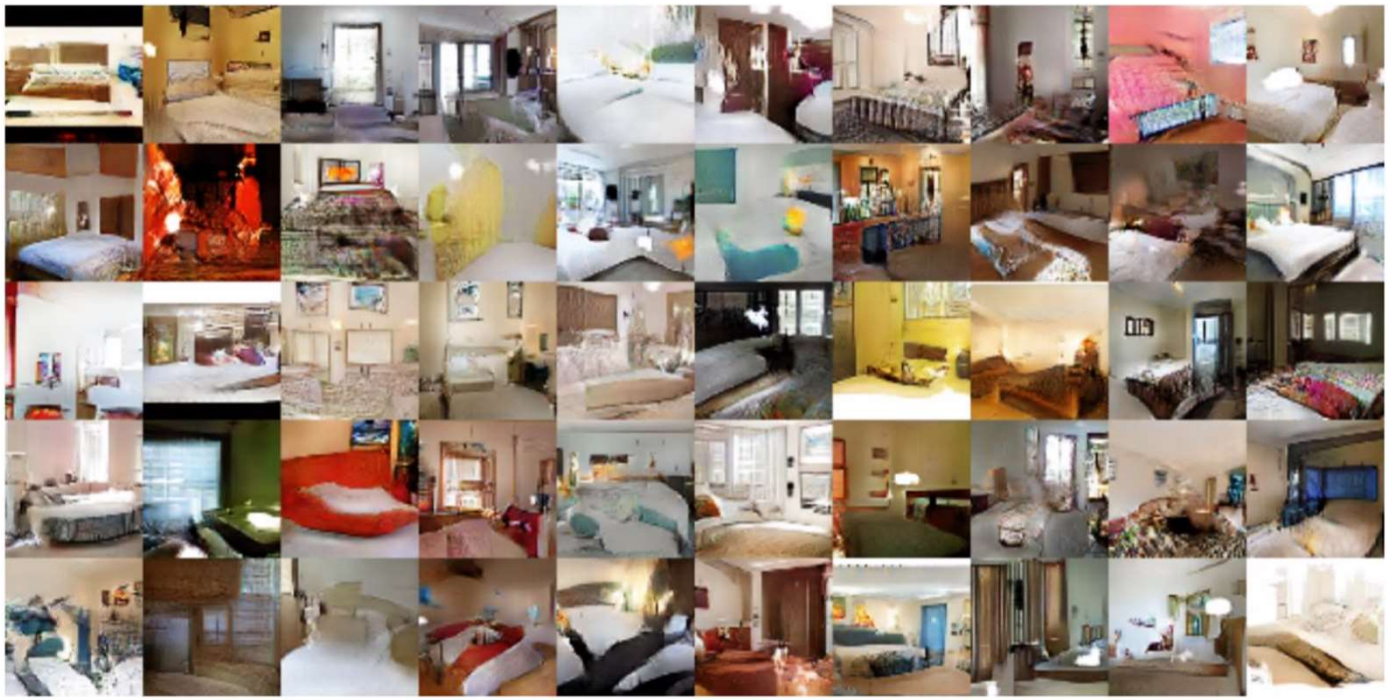
- Since ~2017 extreme number of papers on GANS
- Tricks with loss function for faster learning
  - Wasserstein GANs

- Using better architectures
  - CNN: Deep Convolutional GANs “DC-GAN”

- Clever Design (the bastelbrothers)
  - Semi-Supervised learning SGAN
  - CGANs
  - pix2pix
  - cycle GANs

Using CNN: DC-GAN

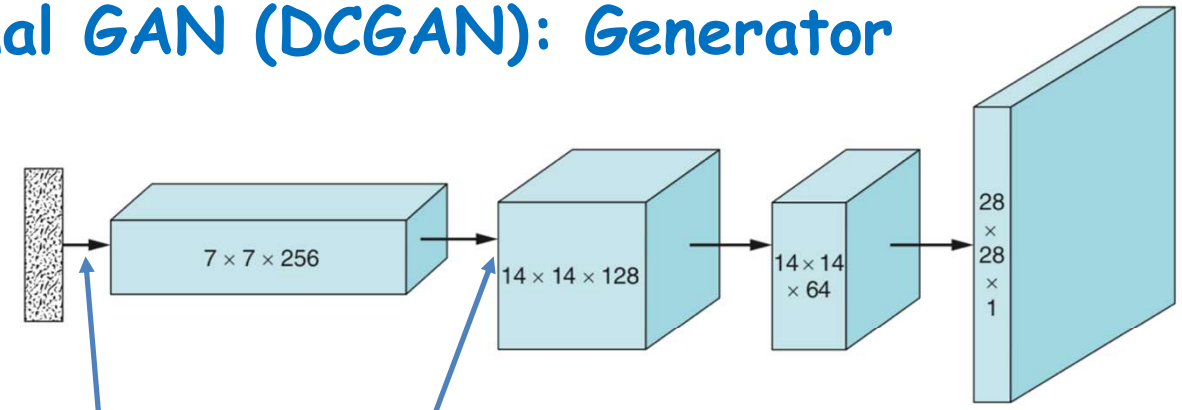
- Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016 DCGAN



- Samples trained on bedrooms
- Toy example from https://www.manning.com/books/gans-in-action

Deep Convolutional GAN (DCGAN): Generator

Only Generator and Discriminator changes



Listing 4.3 DCGA

```
def build_generator(z_dim):

    model = Sequential()

    model.add(Dense(256 * 7 * 7, input_dim=z_dim))
    model.add(Reshape((7, 7, 256)))

    model.add(Conv2DTranspose(128, kernel_size=3, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(64, kernel_size=3, strides=1, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(1, kernel_size=3, strides=2, padding='same'))

    model.add(Activation('tanh'))

    return model
```

Reshapes input into 7 × 7 × 256 tensor via a fully connected layer

Transposed convolution layer, from 7 × 7 × 256 into 14 × 14 × 128 tensor

Batch normalization

Leaky ReLU activation

Transposed convolution layer, from 14 × 14 × 128 to 14 × 14 × 64 tensor

Batch normalization

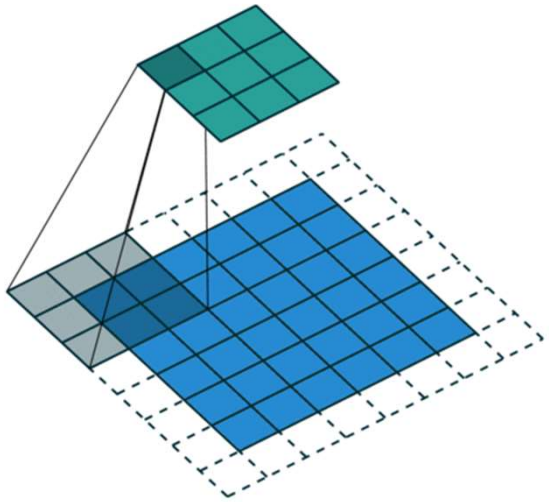
Leaky ReLU activation

Output layer with tanh activation

Transposed convolution layer, from 14 × 14 × 64 to 28 × 28 × 1 tensor

Key Layer: transposed convolutions

- Need to go from say 6x6 to 3x3
- Going from 6x6 to 3x3 can be done with stride (2x2) and padding same



https://github.com/vdumoulin/conv_arithmetic/

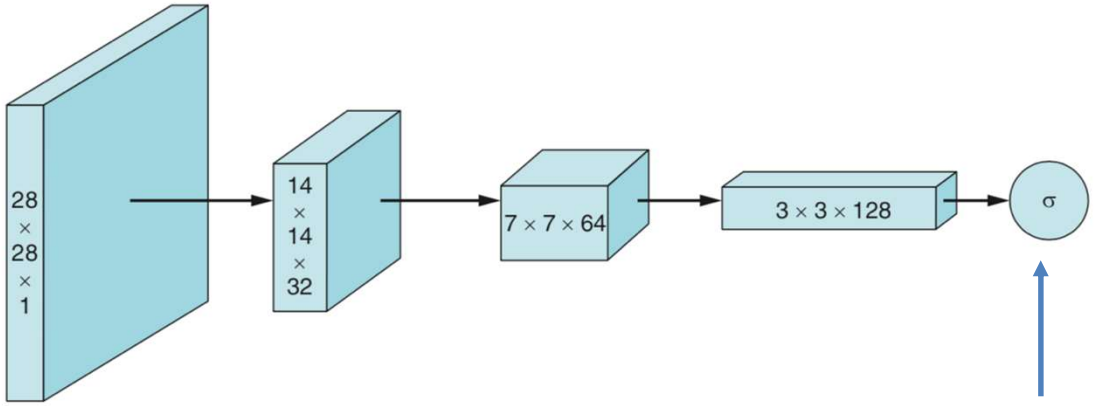
- Convolution is reversible (called Transposed*)
- So `Conv2dTranspose(…, stride=(2,2),…)` goes from 7x7 to 14x14
- Transposed Convolution also in many other architectures like U-Net

*Transposed refers to the fact that convolution can be seen as vector time matrix operation of flattend feature maps. See https://arxiv.org/abs/1603.07285 for more.

Deep Convolutional GAN (DCGAN): Discriminator

Listing 4.4 DCGAN Discriminator

```
def build_discriminator(img_shape):

    model = Sequential()

    model.add(                                  <- Convolutional lay
        Conv2D(32,                                 × 1 into 14 × 1
               kernel_size=3,
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(LeakyReLU(alpha=0.01))            <- Leaky ReLU activation

    model.add(                                  <- Convolutional layer,
        Conv2D(64,                                 from 14 × 14 × 32
               kernel_size=3,                      into 7 × 7 × 64 tensor
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(BatchNormalization())             <- Batch normalization

    model.add(LeakyReLU(alpha=0.01))            <- Leaky ReLU activation

    model.add(                                  <- Convolutional layer, from
        Conv2D(128,                                7 × 7 × 64 tensor into
               kernel_size=3,                      3 × 3 × 128 tensor
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(BatchNormalization())             <- Batch normalization

    model.add(LeakyReLU(alpha=0.01))            <- Leaky ReLU activation

    model.add(Flatten())                        <- Output layer with sigmoid activation
    model.add(Dense(1, activation='sigmoid'))

    return model
```



Output ∈ [0,1]

Sampling from generator

DCGAN

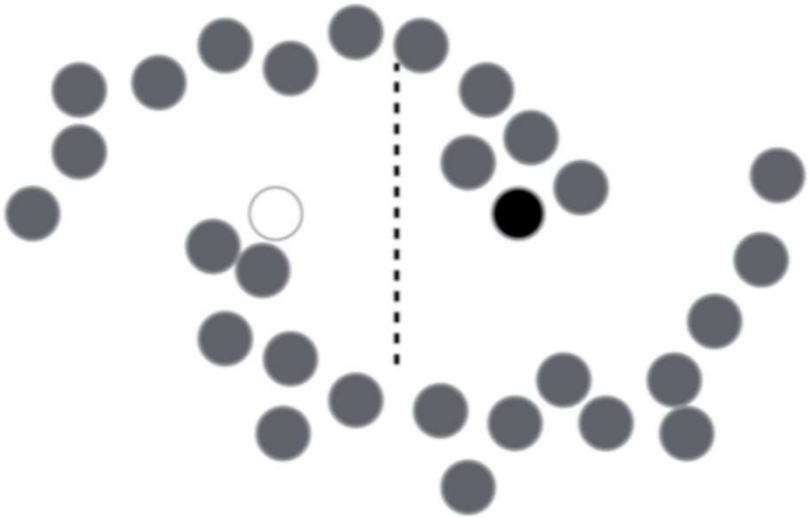
Fully Connected

Quite some enhancement!

Try yourself: https://github.com/GANs-in-Action/gans-in-action/tree/master/chapter-4

Semi Supervised Learning SGAN

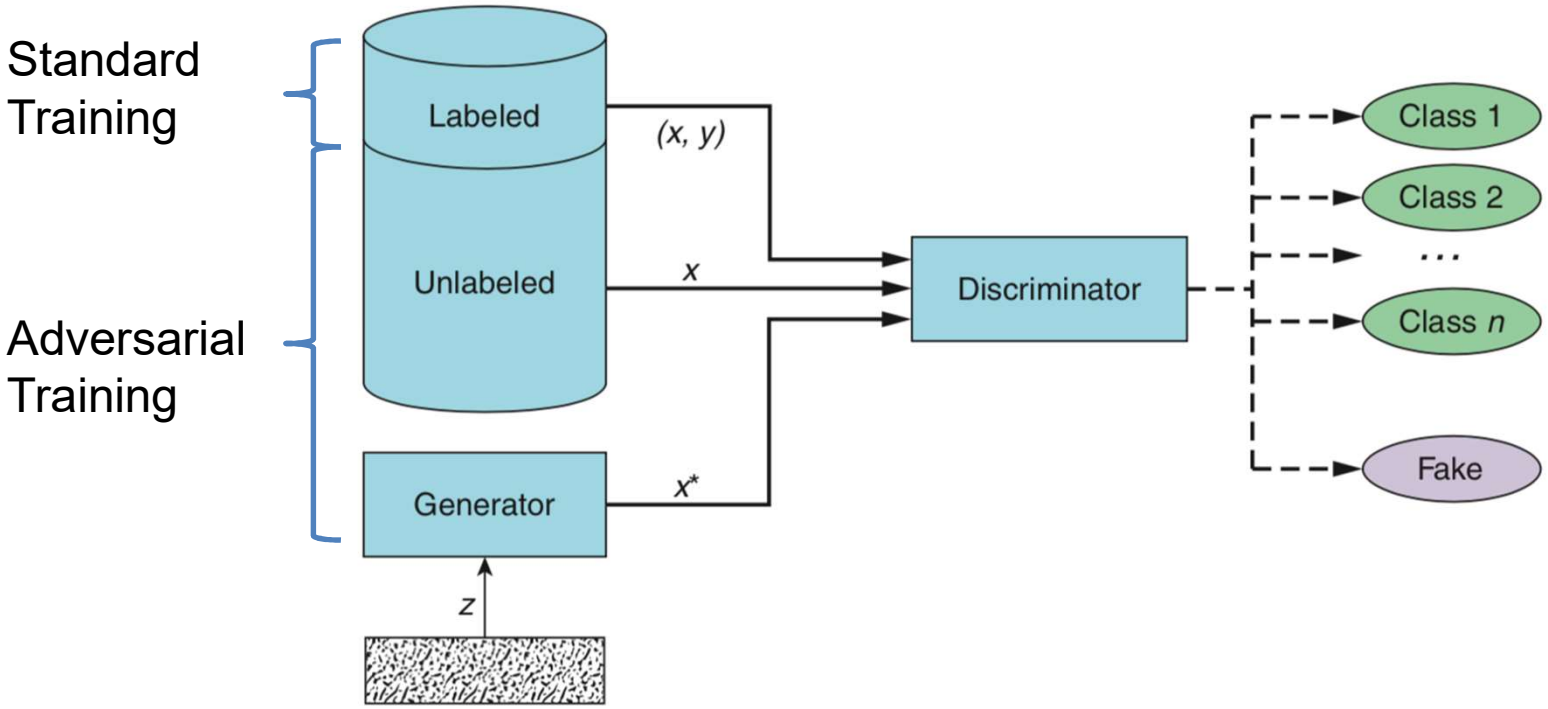
- See introduction ML



Quelle: TapaniRaiko

- Quite important in practice:
  - The internet is full of images but few are labeled.
  - Medical image analysis cheap to get images expensive to consult doctor

Principle SGAN



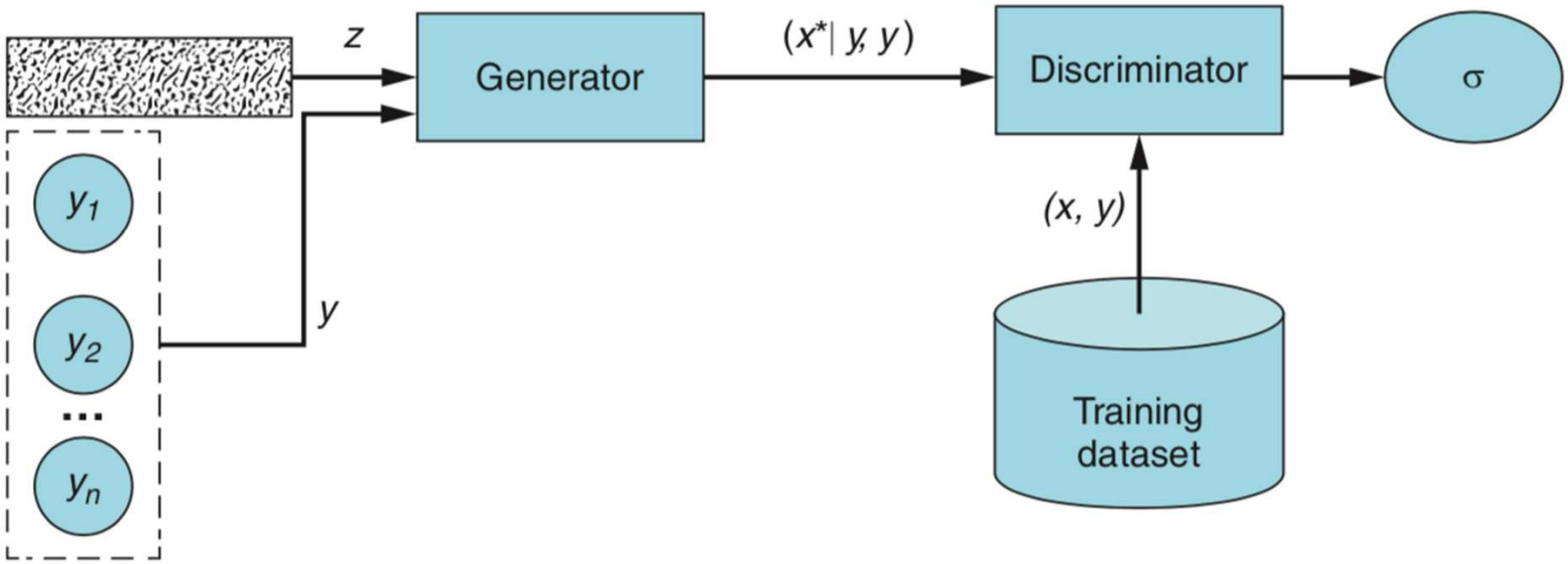
Training scheme for Discriminator (provide two tasks for discriminator)

- Real Images (Unlabeled) / Fake images (G) Fake output loss: binary_cross_entropy
- Labeled different classes Class 1 ... n loss: cross_entropy

Both task train weights e.g. convolutional kernel in the discriminator.

Images taken from https://www.manning.com/books/gans-in-action

Further development of GANs: Conditional



Discriminator need to tell apart

- Real (x, y) from Fake $(x^*_{|y}$,y) with $x^*_{|y}$ is generated conditional on y
- Correct only if class and label matches

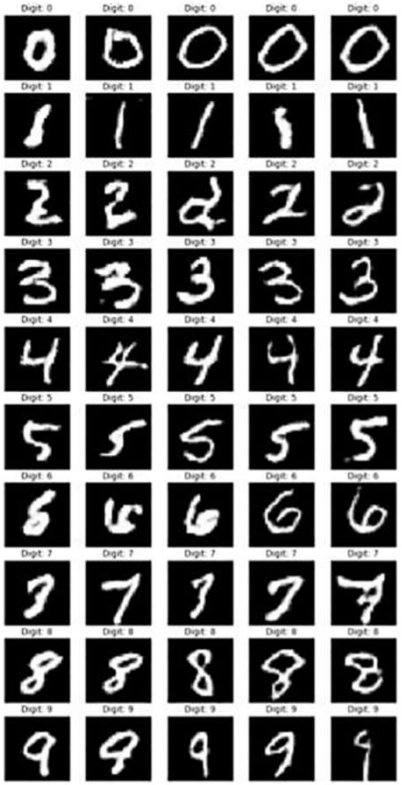
Needs pairs of real and fake

Images taken from https://www.manning.com/books/gans-in-action

Further development of GANs: Conditional

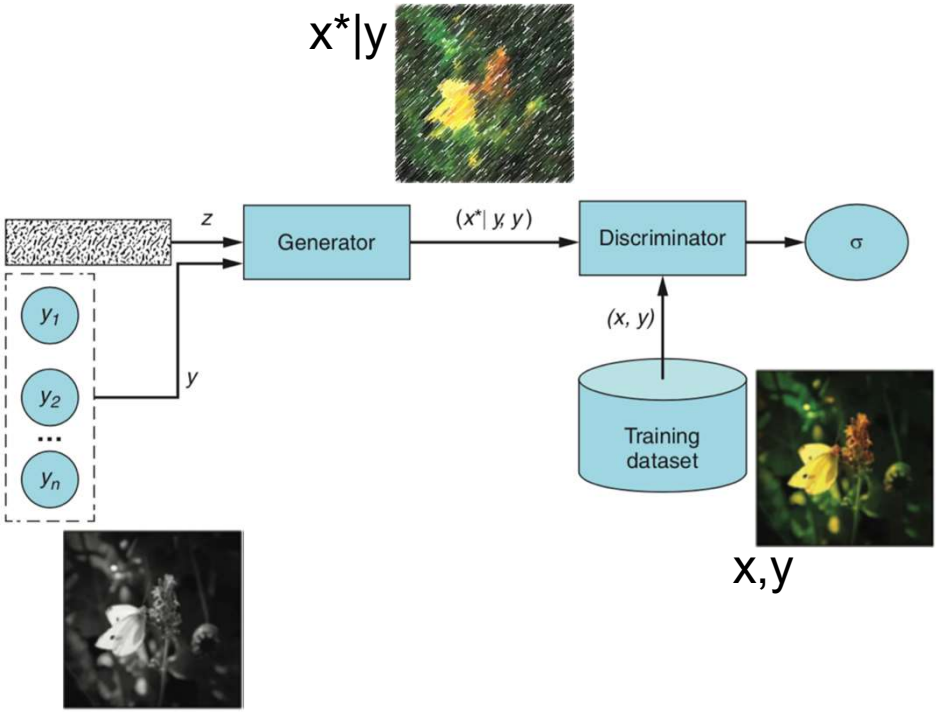
Conditional GAN

Generate Based on Labels



Images taken from https://www.manning.com/books/gans-in-action

Further development of GANs: Conditional pix2pix



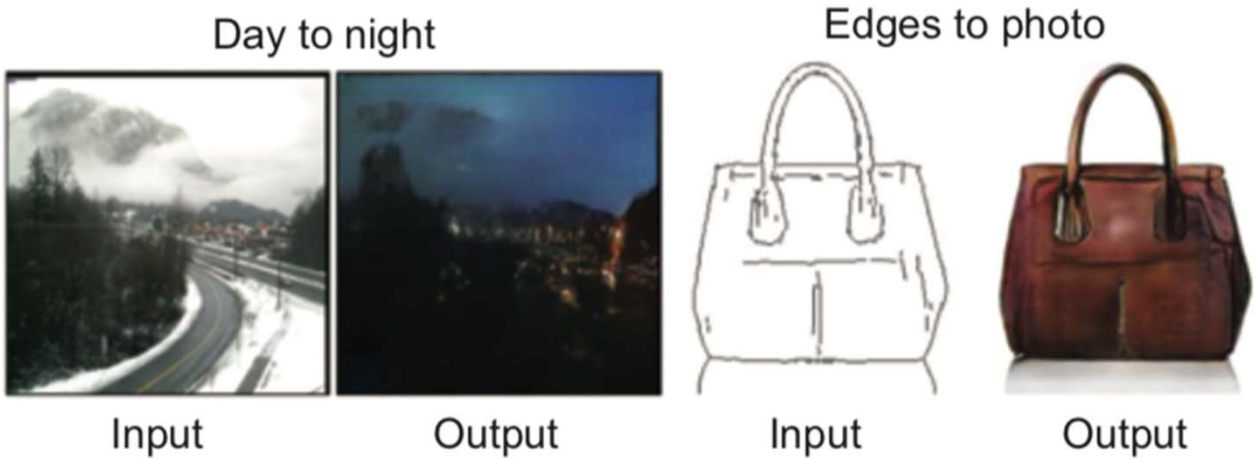
Needs pairs.
Example take color image x
create bw image

Zack fertig you have a pair(x,y)

Generator learns to colorize

Images taken from https://www.manning.com/books/gans-in-action

Further development of GANs: Conditional pix2pix

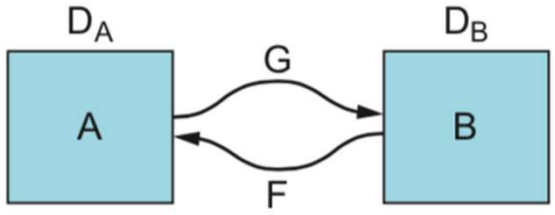


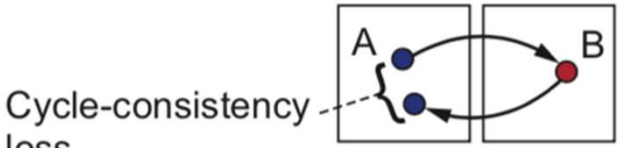
Need pairs, for some tasks this is possible (bw, edgedetection)

Images taken from https://www.manning.com/books/gans-in-action

Further development: cycle GANs

- Principle idea go from one domain (A e.g. zebras) to another (B e.g. horses) and back. **Two Generators** G and F. You should end where you started (cycle consistency)

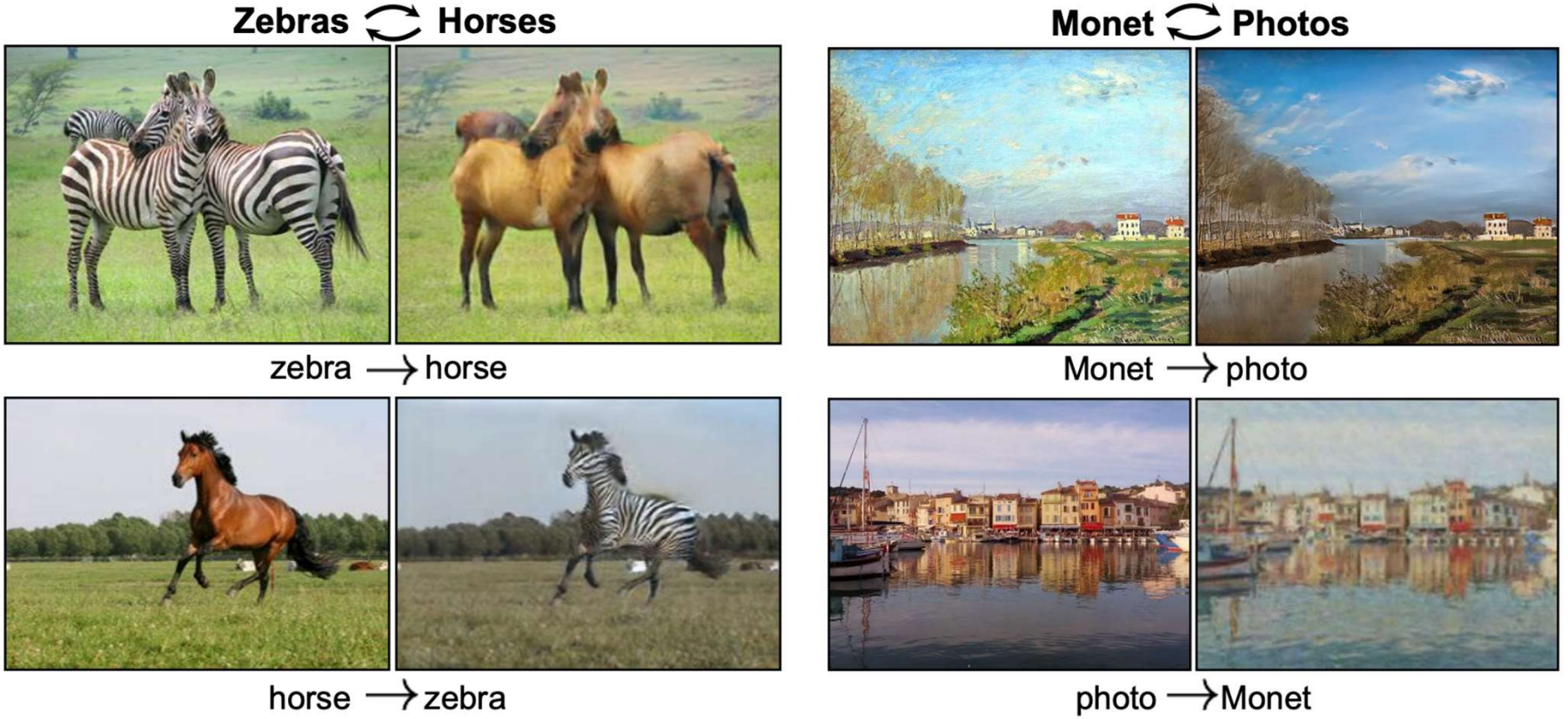




- Also discriminators telling apart if image is from B or generated from A
- Advanced training scheme

Images taken from https://www.manning.com/books/gans-in-action
https://arxiv.org/pdf/1703.10593.pdf

Results



See https://junyanz.github.io/CycleGAN/ for funny videos

Pros and cons

GANs

Don't work with an explicit density function
Take game-theoretic approach: learn to generate from training distribution through 2-player game

Pros:
- Beautiful, state-of-the-art samples!

Cons:
- Trickier / more unstable to train
- Can't solve inference queries such as p(x), p(z|x)

Active areas of research:
- Better loss functions, more stable training (Wasserstein GAN, LSGAN, many others)
- Conditional GANs, GANs for all kinds of applications

Generative models currently (2019) on vogue

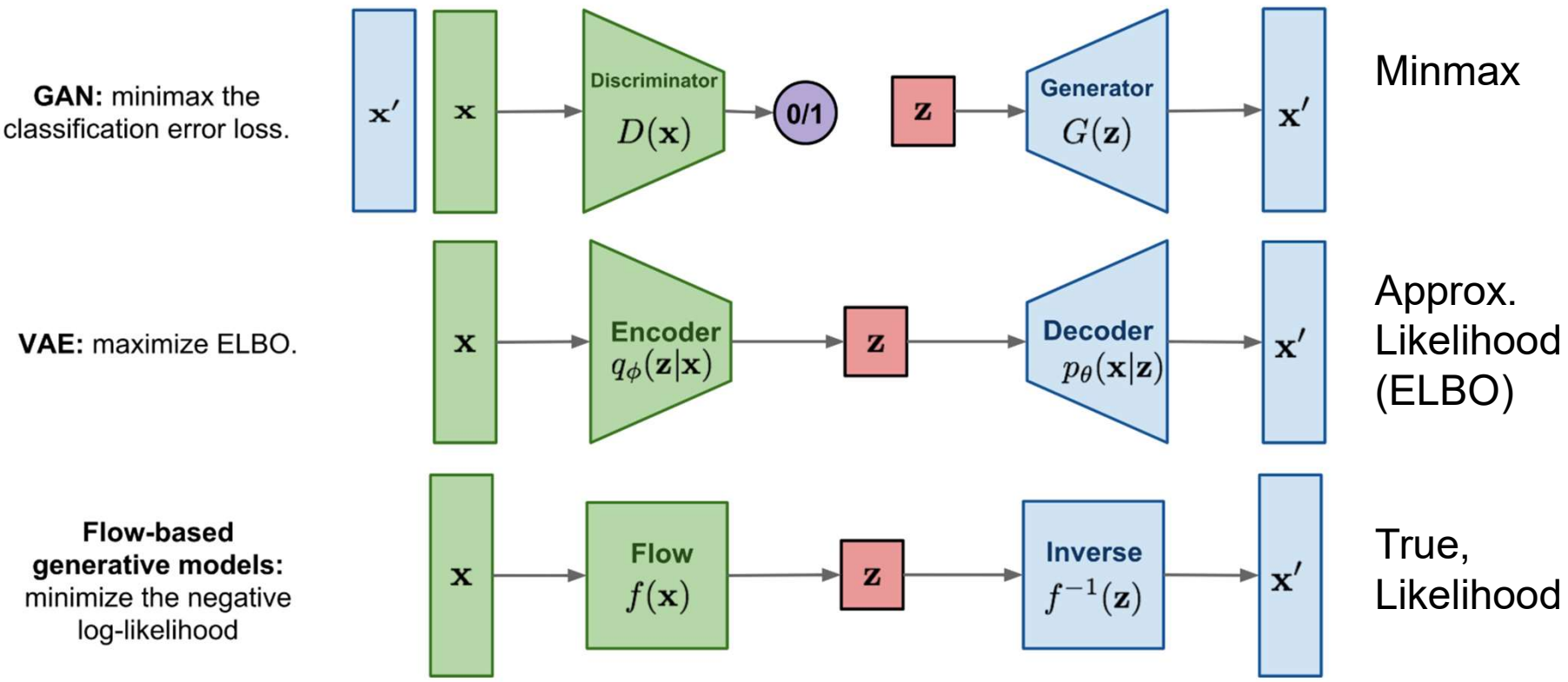


Image (modified) from: https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html

Generative models on vogue

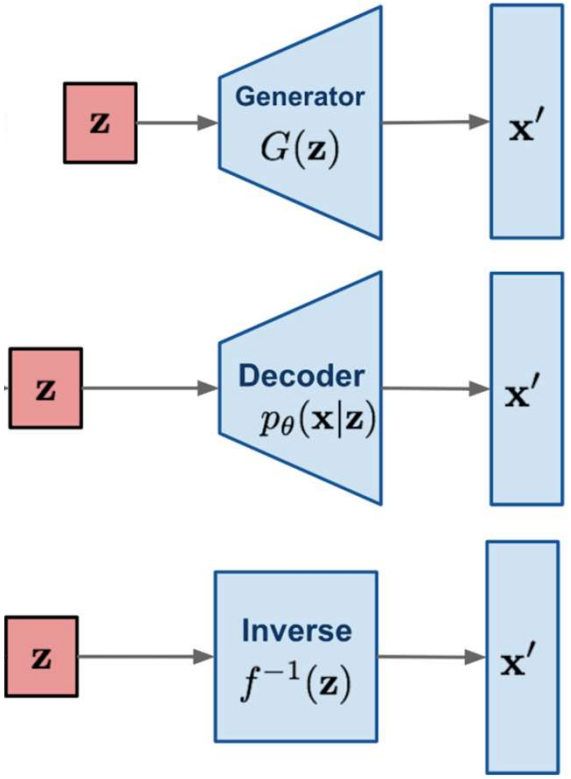
GAN: minimax the classification error loss.

VAE: maximize ELBO.

Flow-based generative models: minimize the negative log-likelihood

Different Training,
Same generative process

Z→X



Minmax

Approx. Likelihood (ELBO)

True, Likelihood

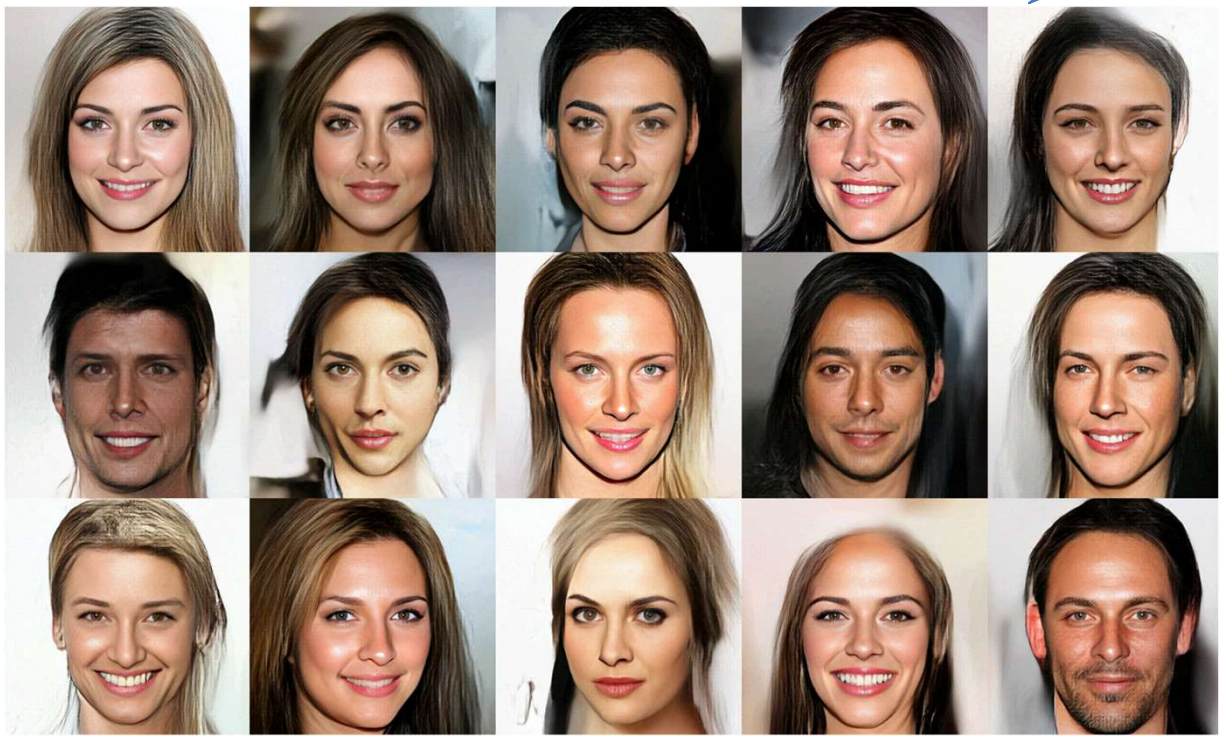
Image (modified) from: https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html

Flow Based Models
-- The crown of generation (IMHO)

Go with the flow

An introduction to normalizing flows

These people are not real they are generated samples using NF



A bit of Motivation

- A the End of the lecture, you can create and understand something like:

- Look at the intermediate pictures, they look real.
- Persons no celebrities (not part of celebA-HQ used for training)

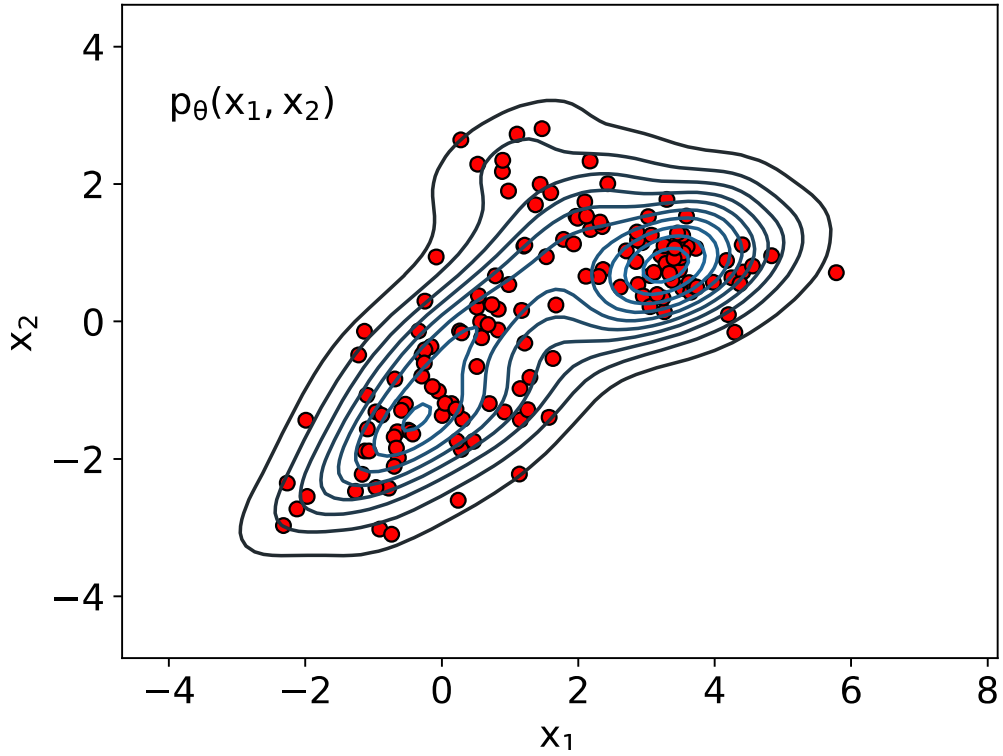
Outline

- Classification and motivate NF
  - Density Estimation
  - Generative Models
  - Need for flexible distributions
- Change of Variables
- Using networks to control flows
  - RealNVP
- Glow for image data

- Demo code is in
  - https://github.com/tensorchiefs/dl_book/tree/master/chapter_06

Normalizing Flows

- An novel method of parametric density estimation



- Example of parametric density estimation 2-D Gaussians with μ and Σ

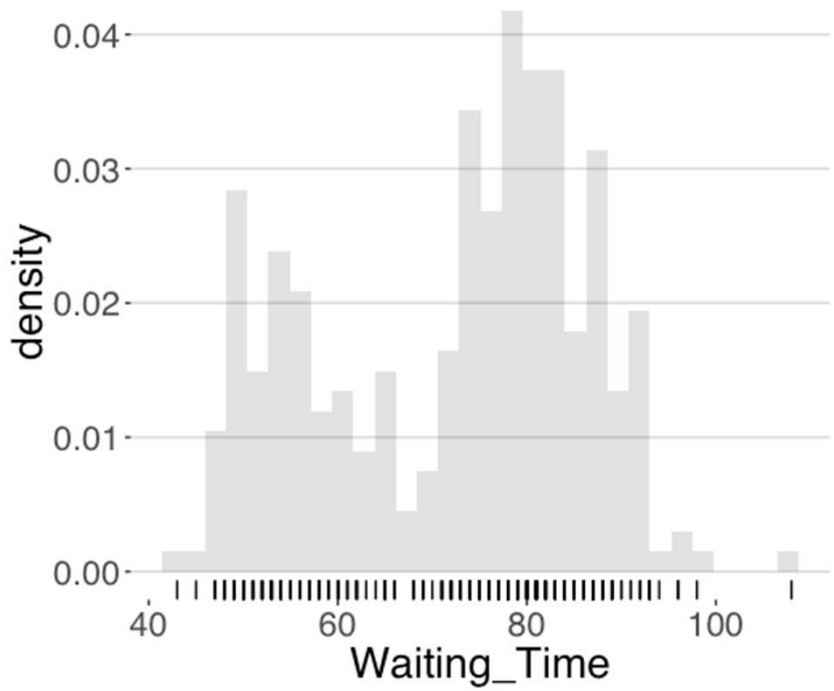
Theory: Name Some Distributions

- Gaussian
- Uniform
- Weibull
- Binomial
- Log-Normal

These are the distributions we have in our Toolbox.

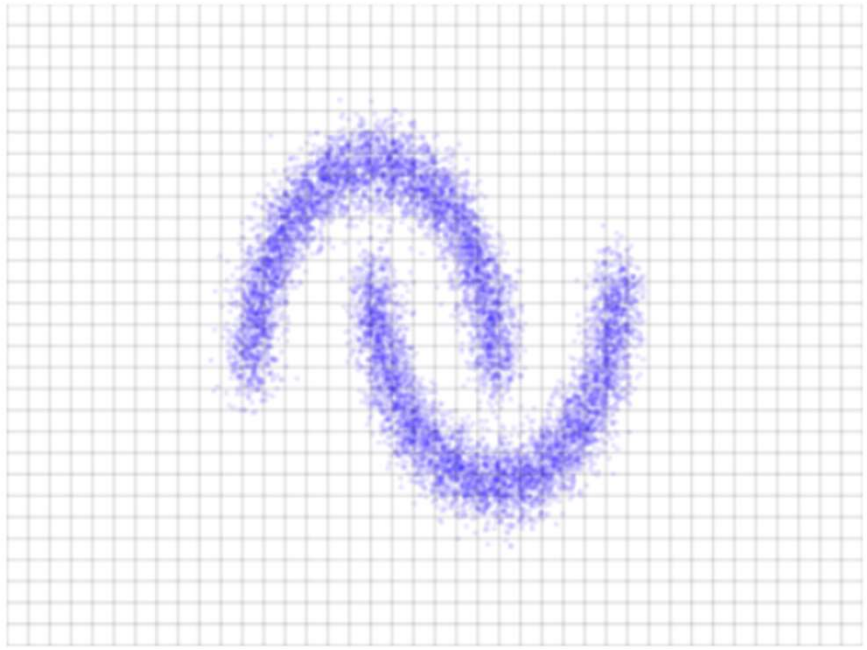
Is the reality like this?

Reality: Data (1-D)



What distribution can you use?

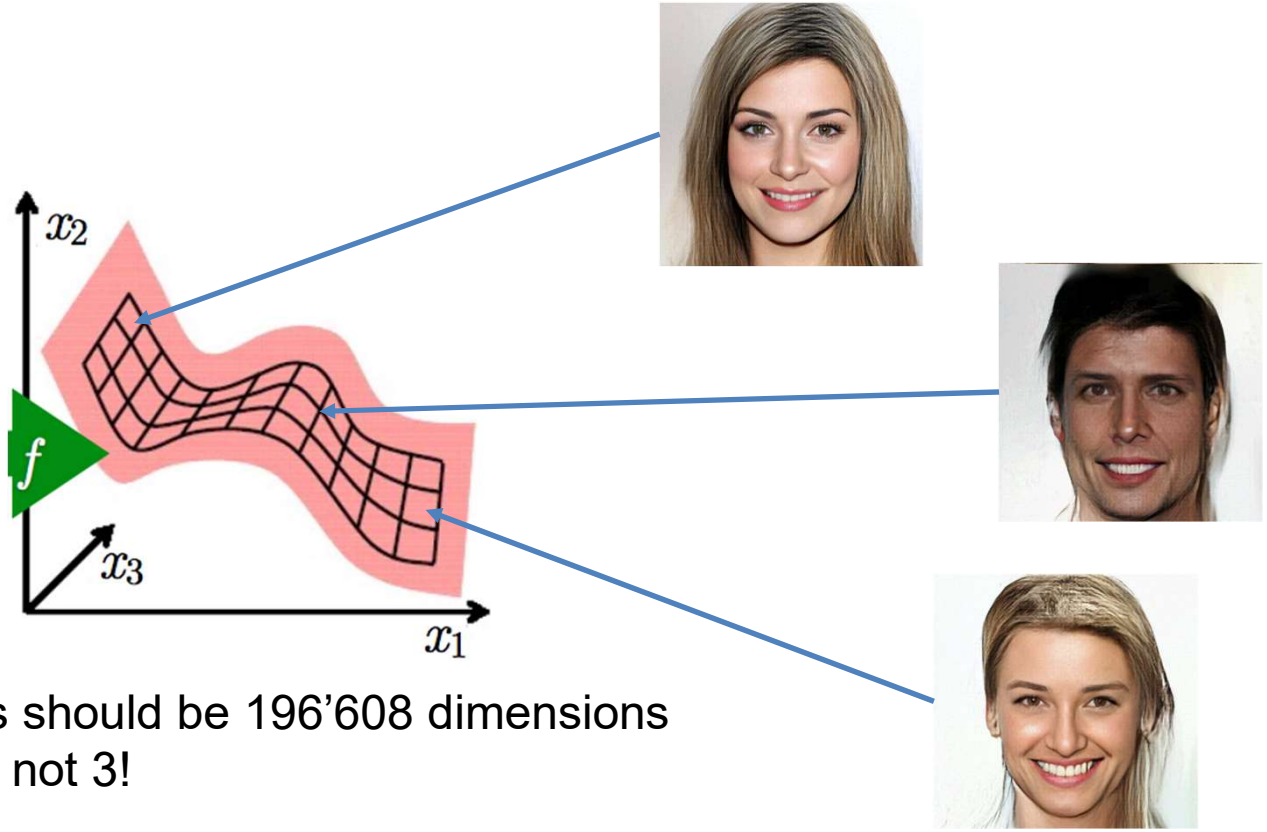
Reality: Data (2-D)



What distribution can you use?

Reality: Data (256x256x3=196'608 Dimensions)

3 data points **sampled** from the high dimensional distribution



This should be 196'608 dimensions and not 3!

What distribution can you use?

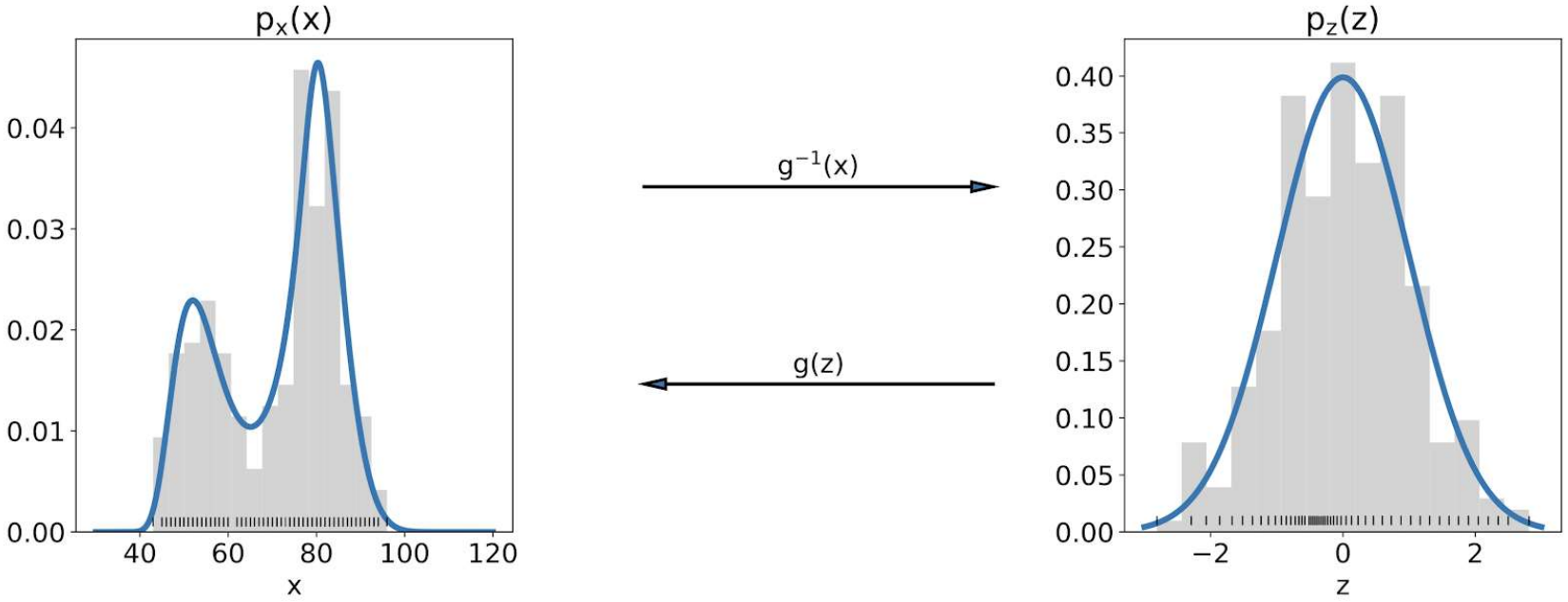
Aproches for Density Estimation task, we want $p_\theta(X)$:

- For easy cases fit normal “estimate mean and variance”
  - Limited to simple distributions
- Mixtures of simple Distributions such as Gaussian
  - Limited to fairly simple distribution
- Kernel Density estimation / Histograms
  - Non-Parametric, low dimensions (non-sparse)
- Copulas (ideas from finance)
  - Limited to some 10 or 100 dimensions
- MCMC
  - Allows to **sample** from complicated distributions
- GANs (only have an *implicit* estimation can sample from $p(X)$)
- VAE (only have an approximation to $p(X)$)
  - $\log\big(p(x)\big) = L^v + \cancel{D_{kl}(q(z|x)||p(z|x))}$ the KL-Term is disregarded
- **Normalizing Flows**

Main Idea of Normalizing Flows

Data x~strange_function in $\mathbb{R}^1$

Transformed function $z_1 \sim N(0,1)$



Idea: learn an *invertible* transformation to simple function usually Gaussian $N(0,1)$

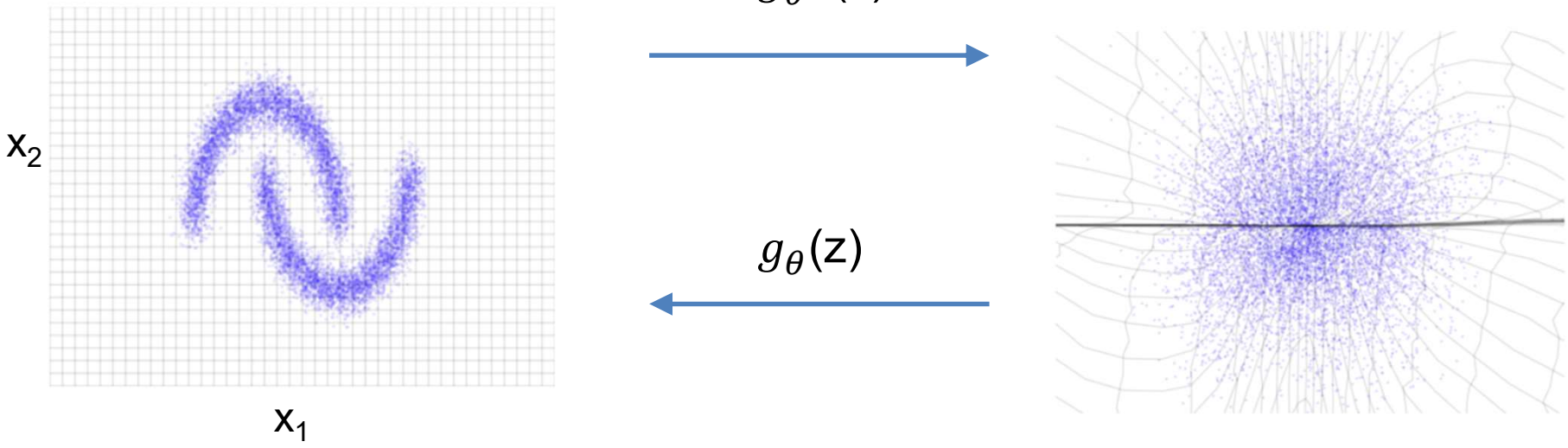
- Sampling from $p_x(x)$: sample $z^* \sim p_z(z)$ then transform it via g_θ(z)
- Density of *x**: calculate z*= $g_\theta^{-1}(x^*)$ and evaluate $N(z^*; 0,1)$

There is no fixed naming convention. Also in the direction of g(z) and $g^{-1}(x)$.

Main Idea of Normalizing Flows

Data x~strange_function in $\mathbb{R}^2$

Transformed Data $z_1, z_2 \sim N(0,1)$



pdf $p_x(x)$

pdf $p_z(z)$

Idea: learn an *invertible* transformation to simple function usually Gaussian $N(0,1)$

- Sampling from $p_x(x)$: sample $z^* \sim p_z(z)$ then transform it via $g_\theta(z)$
- Density of x^*: calculate $z^* = g_\theta^{-1}(x^*)$ and evaluate $N(z^*; 0,1)$

Image Credit: RealNVP

Main Idea of Normalizing Flows

Data x~strange_function in $\mathbb{R}^{196608}$

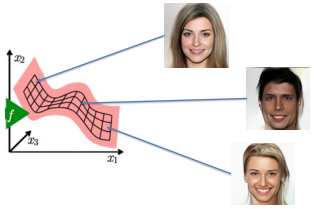
g_θ^{-1}(x)

x_1, x_2, x_{196608}~strange_function

$z_1, z_2, \ldots, z_{196608} \sim N(0,1)$

With many correlations

g_θ(z)



pdf p(x)

pdf $\mathrm{p_z}(z)$

Idea: learn an *invertible* transformation to simple function usually Gaussian $N(0,1)$

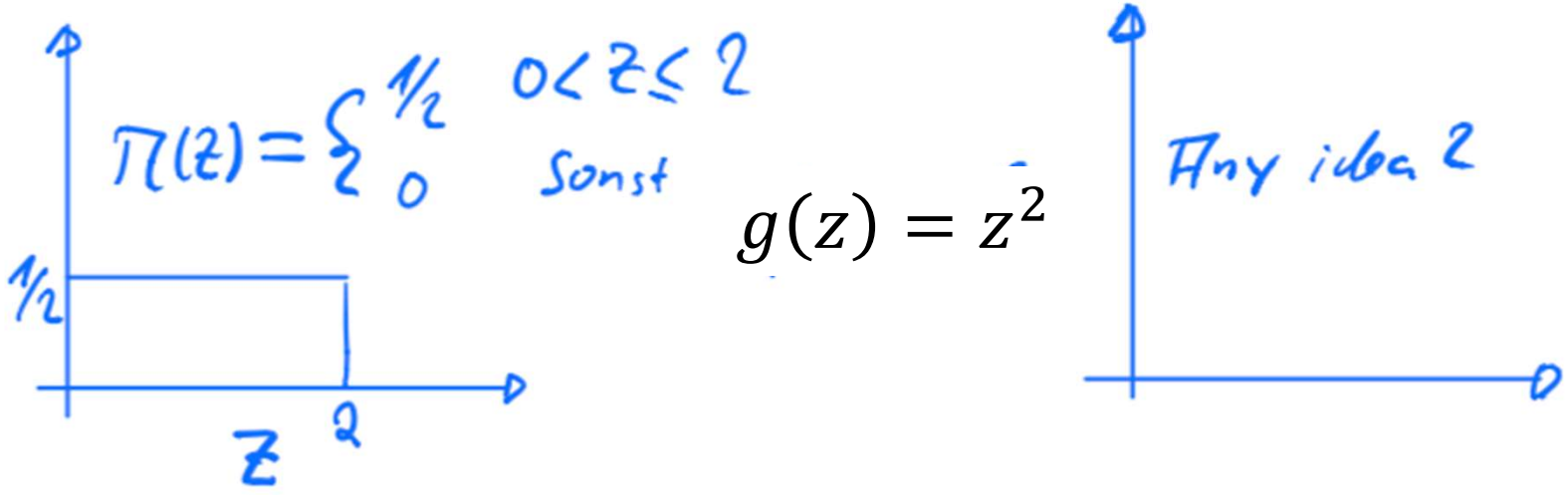
- Sampling from $p_x(x)$: sample $z^* \sim p_z(z)$ then transform it via g_θ(z)
- Density of *x**: calculate z*= $g_\theta^{-1}(x^*)$ and evaluate $N(z^*; 0,1)$

Transformation of Variables -- Some math

Simple Transformation

- Say you have z~$Uniform(0,2)$
- $g(z) = z^2$

```
N = 10000
d = tfd.Uniform(low=0, high=2)
zs = d.sample(N)
x = zs**2
```



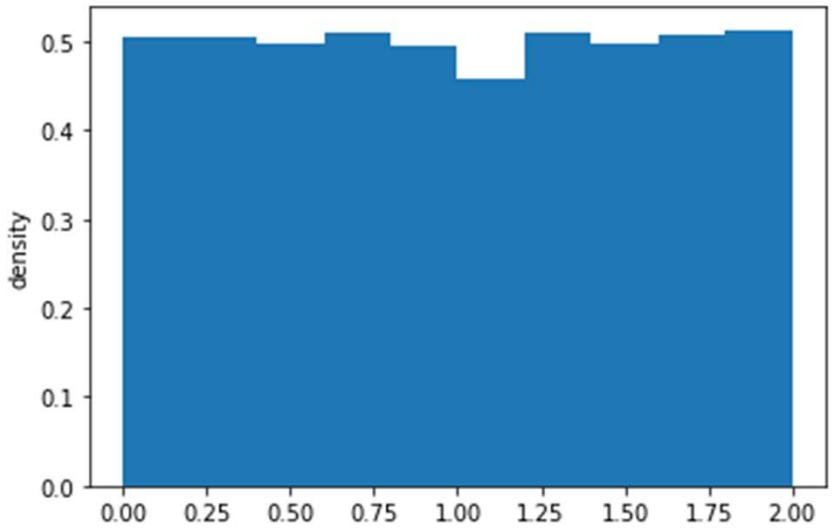
Try to come up with an answer, how is z distributed?

Try it

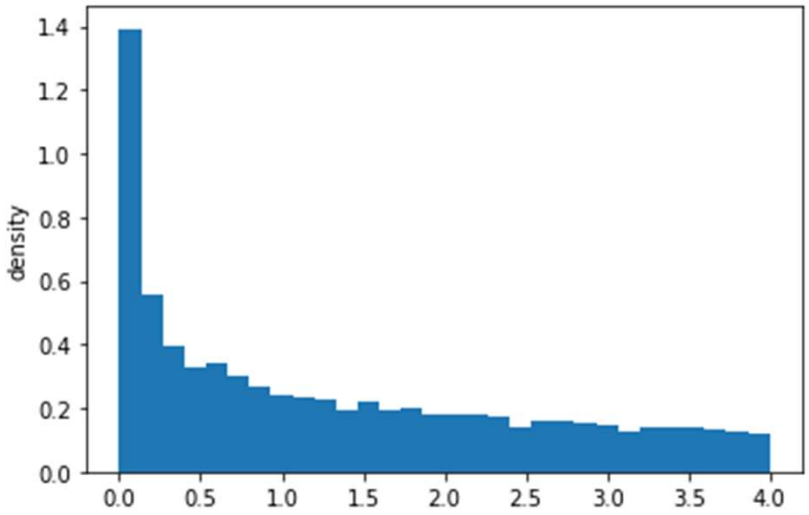
```
N = 10000
d = tfd.Uniform(low=0, high=2)
zs = d.sample(N)
```

```
x = zs**2
```

hist zs

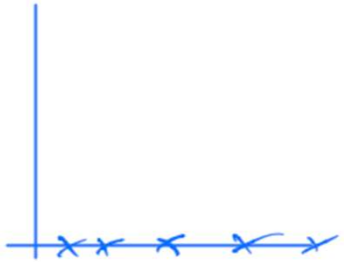


hist zs**2

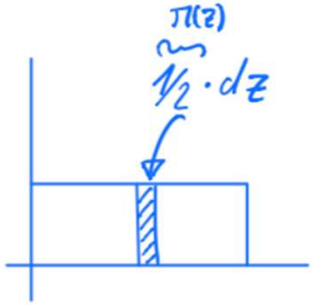


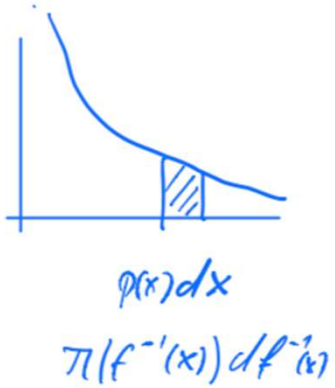
What happened?
Probability Mass needs to be conserved

Think of samples



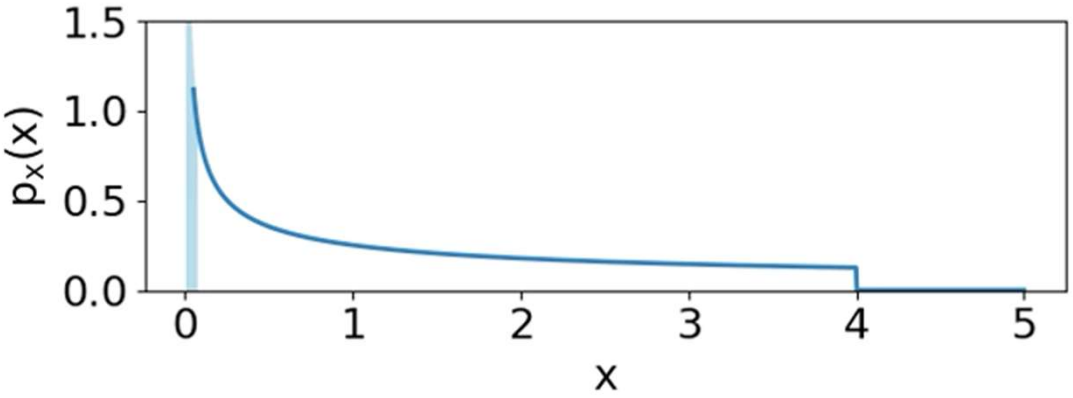
Think of mass needs to be conserved

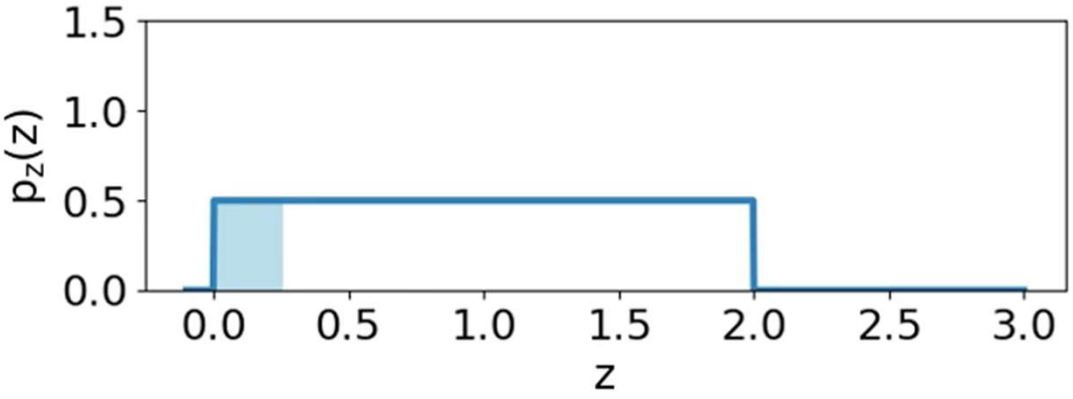




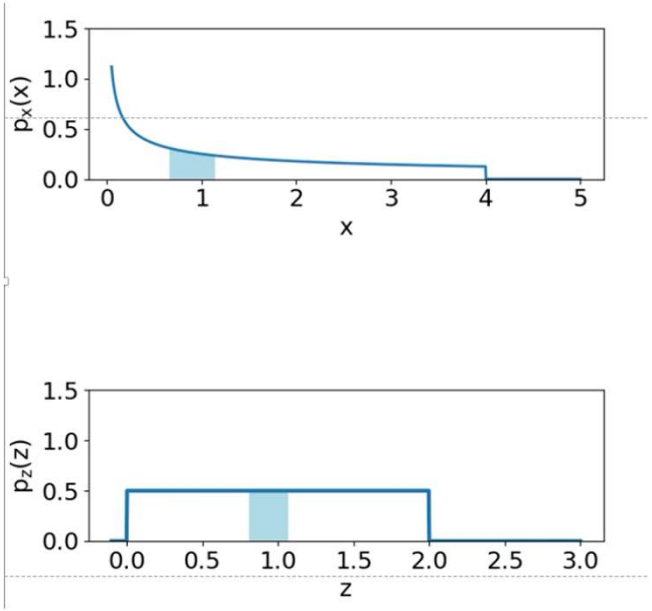
$$\pi(z)\,dz = p(x)\,dx$$

Conservation of probability





1-D



For small value dz and dx :

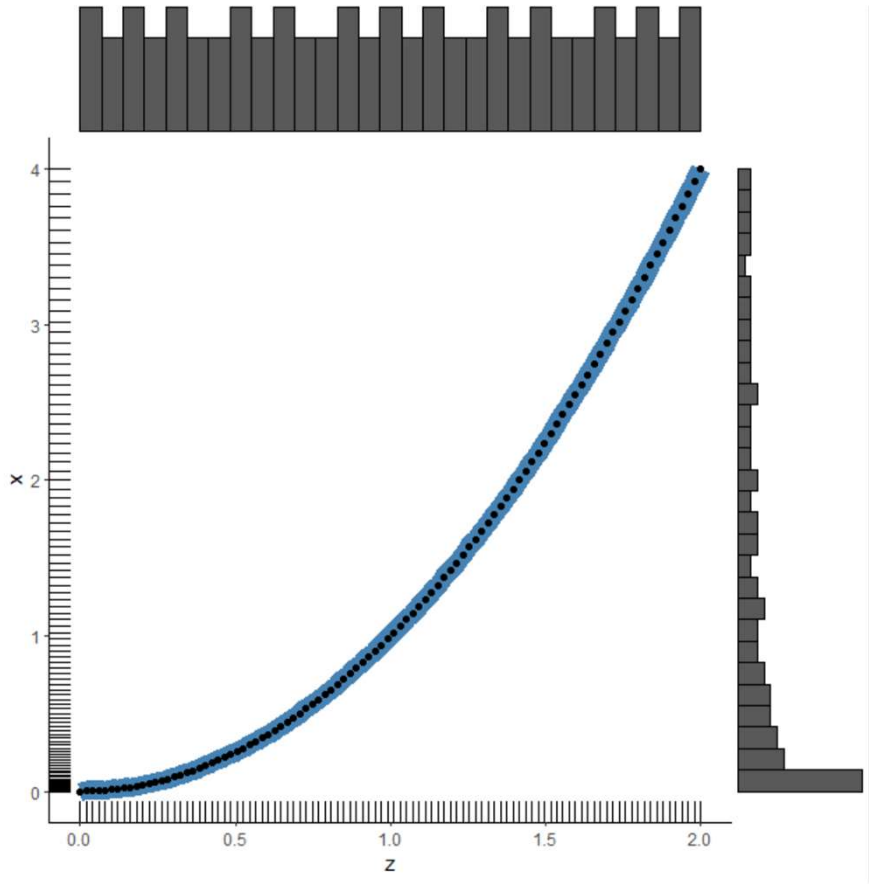
$$p_z(z) \cdot |dz| = p_x(x) \cdot |dx|$$

$$p_x(x) = p_z(z) \cdot |\frac{dz}{dx}| = p_z(z) \cdot |\frac{dx}{dz}|^{-1} = p_z(z) \cdot |\frac{dg(z)}{dz}|^{-1} = p_z(g^{-1}(x)) \cdot |g'(g^{-1}(x))|^{-1}$$

$$p_x(x) = p_z(z) \cdot |g'(z)|^{-1} = p_z(g^{-1}(x)) \cdot |g'(g^{-1}(x))|^{-1}$$

Example See Blackboard

Annother View



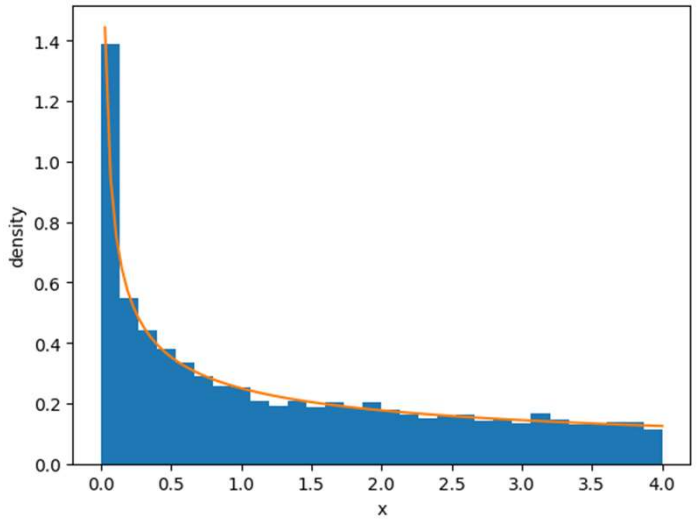
Definition in TFP the transformation is a Bijector

Listing 6.tfb2: The first bijector

```
tfb = tfp.bijectors
g = tfb.Square() #A
g.forward(2.0)   #B
g.inverse(4.0)   #C

#A This is a simple bijector going from z → z**2
#B Yields 4
#C Yields 2
```

Listing 6.tfb3: The simple example in TFP

```
# 10          20        30        40        50          55
#12345678901234567890123456789012345678901234567890123456789012345
g = tfb.Square() #A
db = tfd.Uniform(0.0,2.0) #A2
mydist = tfd.TransformedDistribution( #B
    distribution=db, bijector=g)

xs = np.linspace(0.001, 5,1000)
px = mydist.prob(xs) #C
```



https://github.com/tensorchiefs/dl_book/blob/master/chapter_06/nb_ch06_03.ipynb

Learning to flow

- How probable (well density) is a data point x_i (use $z_i = g^{-1}(x_i)$)

$$p_x(x_i) = p_z(z_i) \cdot |g'(z_i)|^{-1}$$

- All Data points

$$\prod_{i=1}^{n} p_x(x_i)$$

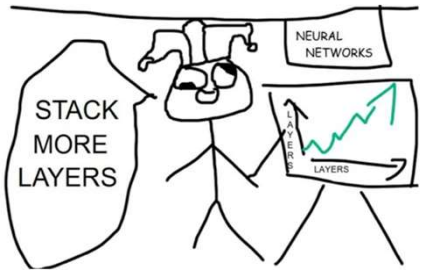
Tune the parameter(s) θ of the model M so that (observed) data is most likely!

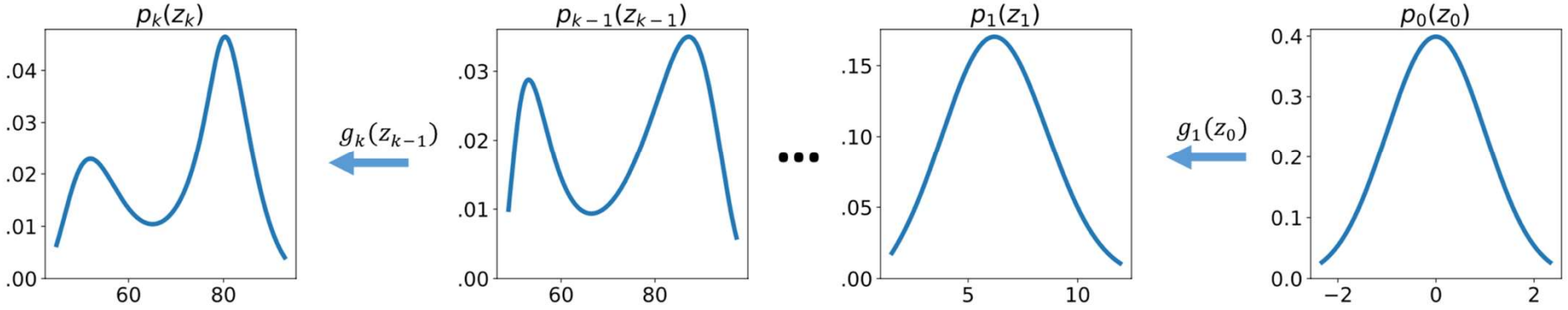
- Simple example affine linear

$$g_{(a,b)}(z) = a \cdot z + b$$

- https://github.com/tensorchiefs/dl_book/blob/master/chapter_06/nb_ch06_03.ipynb

Chaining





z0->z1->z2=x

$$p_{z_1}(z_1) = p_{z_0}(z_0) \cdot |g_1'(z_0)|^{-1}$$

$$p_{z_2}(z_2) = p_{z_1}(z_1) \cdot |g_2'(z_1)|^{-1}$$

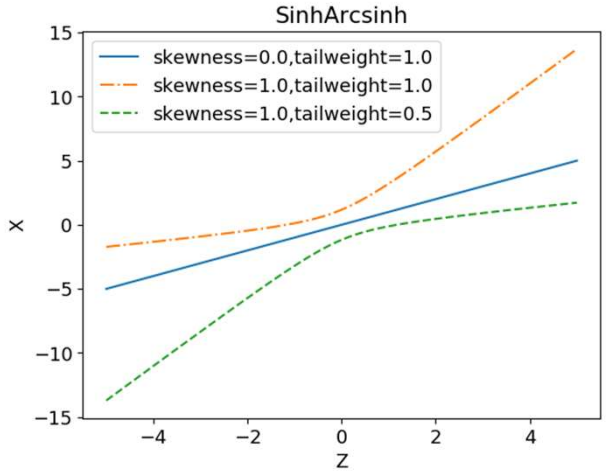
$$p_{z_2}(z_2) = p_{z_0}(z_0) \cdot |g_1'(z_0)|^{-1} \cdot |g_2'(z_1)|^{-1}$$

$$log(p_{z_2}(z_2)) = log(p_{z_0}(z_0)) - log(|g_1'(z_0)|) - log(|g_2'(z_1)|)$$

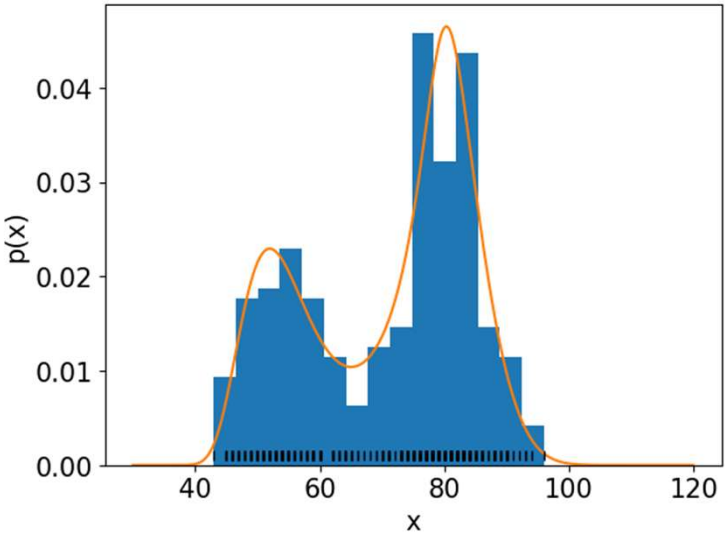
$$log(p_x(x)) = p_{z_0}(z_0) - \sum_{i=1}^{k} log(|\frac{dg_i(z_{i-1})}{dz_{i-1}}|)$$

Practical example

- Need non-linearity if using simple scale and shift



- Geyser Data



Going to higher dimensions

Transformation in high dimensions

$$\mathrm{g}(z) = \begin{pmatrix} g_1(z_1, z_2, z_3) \\ g_2(z_1, z_2, z_3) \\ g_3(z_1, z_2, z_3) \end{pmatrix}$$

$$\frac{\partial g(z)}{\partial z} = \begin{pmatrix} \frac{\partial g_1(z_1, z_2, z_3)}{\partial z_1} & \frac{\partial g_1(z_1, z_2, z_3)}{\partial z_2} & \frac{\partial g_1(z_1, z_2, z_3)}{\partial z_3} \\ \frac{\partial g_2(z_1, z_2, z_3)}{\partial z_1} & \frac{\partial g_2(z_1, z_2, z_3)}{\partial z_2} & \frac{\partial g_2(z_1, z_2, z_3)}{\partial z_3} \\ \frac{\partial g_3(z_1, z_2, z_3)}{\partial z_1} & \frac{\partial g_3(z_1, z_2, z_3)}{\partial z_2} & \frac{\partial g_3(z_1, z_2, z_3)}{\partial z_3} \end{pmatrix}$$

$$p_x(x) = p_z(z) \cdot \left| det\left(\frac{dg\,(z)}{dz} \right) \right|^{-1}$$

Requirements for the Bijectors

A flow is composed of serval possible different *g's,* the bijectors in TFP language. The following restrictions apply for them

- *g* needs for be invertible (strict requirement)

- Training
  - Fast calculation of $g^{-1}(x)$
  - Fast calculation of Jacobi-Determinant

- Application:
  - Fast calculation of $\mathrm{g}(z)$

Flows with networks

Requirement / Design considerations for networks

- Fast calculation of $g(z),\ g^{-1}(x)$
- Crucial: We need fast calculation of Jacobi Matrix
  - $\left|\det\left(\frac{\partial g_i(z)}{\partial z_j}\right)\right|^{-1}$

$$\begin{pmatrix} \frac{\partial g_1(z)}{\partial z_1} & \frac{\partial g_1(z)}{\partial z_2} & \frac{\partial g_1(z)}{\partial z_3} \\ \frac{\partial g_2(z)}{\partial z_1} & \frac{\partial g_2(z)}{\partial z_2} & \frac{\partial g_2(z)}{\partial z_3} \\ \frac{\partial g_3(z)}{\partial z_1} & \frac{\partial g_3(z)}{\partial z_2} & \frac{\partial g_3(z)}{\partial z_3} \end{pmatrix}$$

- Lin. Alg.: The determinant of triangular matrix is product of diagonal terms
  - Want triangular matrix $\frac{\partial g_1(z)}{\partial z_2} \overset{!}{=} 0$
    - ➔ $g_1(z) = f_1(z_1, \cancel{z_2, z_3})$, $g_d(z) = g_1(z_1, \dots, z_d, \cancel{z_{d+1}, z_{d+2,\dots}})$
  - Diagonal terms $\frac{\partial g_2(z)}{\partial z_2}$ easy to be calculated (no network!)

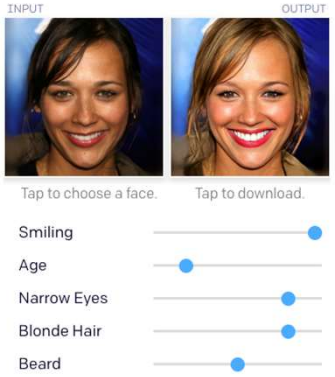
- $\frac{\partial g_2(z)}{\partial z_1}$ no restrictions, can be as complicated as hell (neural network)

Flows using networks

2 Main lines of research

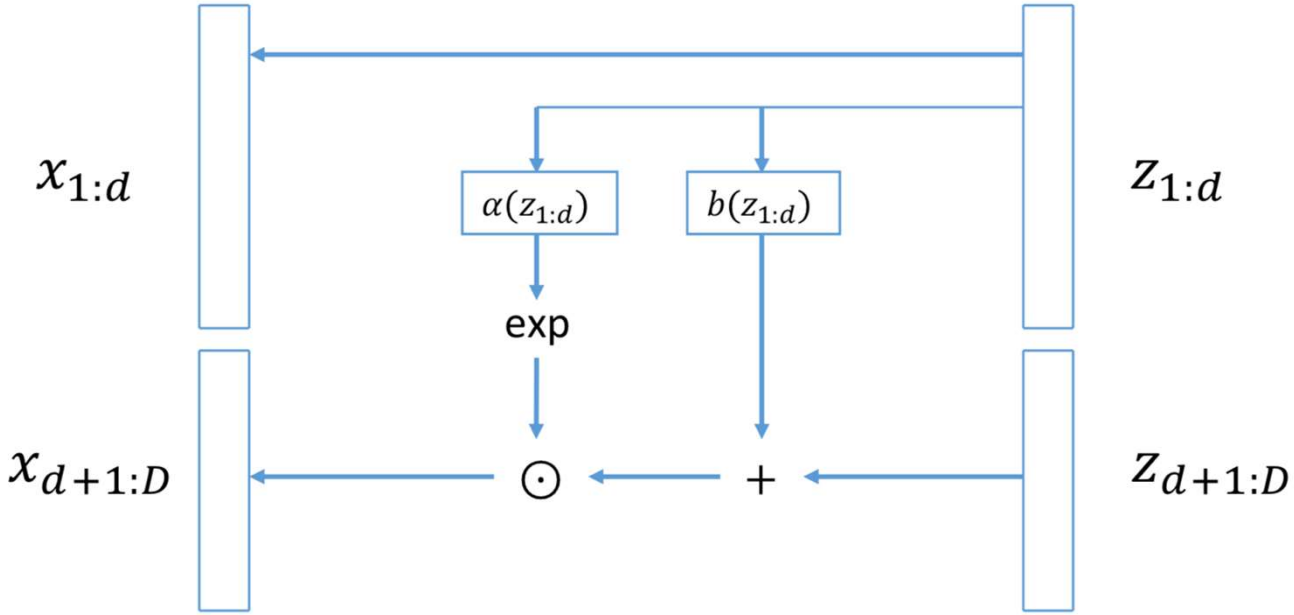
- Guided by autoregressive (AR) models
  - All AR models like Wavenet can be understood as normalizing flows
    - Mask Autoregressive Flow (MAF)
    - Inverse Mask Autoregressive Flow (IMAF)

- Using ‘handcrafted’ network based flows
  - NICE (1410.8516 Dinh, Krueger, Bengio)
  - RealNVP (1605.08803 Dinh, Dickstein, Bengio)
  - Glow (https://arxiv.org/abs/1807.03039 Kingma, Dahriwal)



- Unifying framework (Triangular Maps)
  - SOS paper ICML https://arxiv.org/abs/1905.02325

Simple Solution: RealNVP



Example for D=5 and d=2 $\alpha(z_1{:}\,d)$ and $b(z_1{:}\,d)$ network with D-d output

$$x_1 = g_1(z_1) = z_1$$
$$x_2 = g_2(z_2) = z_2$$
$$x_3 = g_3(z_1, z_2, z_3) = b_3(z_1, z_2) + \exp(\alpha_3(z_1, z_2)) \cdot z_3$$
$$x_4 = g_4(z_1, z_2, z_4) = b_4(z_1, z_2) + \exp(\alpha_4(z_1, z_2)) \cdot z_4$$
$$x_5 = g_5(z_1, z_2, z_5) = b_5(z_1, z_2) + \exp(\alpha_5(z_1, z_2)) \cdot z_5$$

$$z_1 = x_1$$
$$z_2 = x_2$$
$$z_3 = (x_3 - \mu_1(z_1, z_2)) / \exp(\alpha_1(z_1, z_2))$$
$$z_4 = (x_4 - \mu_2(z_1, z_2)) / \exp(\alpha_2(z_1, z_2))$$
$$z_5 = (x_5 - \mu_3(z_1, z_2)) / \exp(\alpha_3(z_1, z_2))$$

Easy invertible ✓, what about the Jacobi-Det?

Simple Solution: RealNVP

$$x_1 = g_1(z_1) = z_1$$
$$x_2 = g_2(z_2) = z_2$$
$$x_3 = g_3(z_1, z_2, z_3) = b_3(z_1, z_2) + \exp(\alpha_3(z_1, z_2)) \cdot z_3$$
$$x_4 = g_4(z_1, z_2, z_4) = b_4(z_1, z_2) + \exp(\alpha_4(z_1, z_2)) \cdot z_4$$
$$x_5 = g_5(z_1, z_2, z_5) = b_5(z_1, z_2) + \exp(\alpha_5(z_1, z_2)) \cdot z_5$$

j

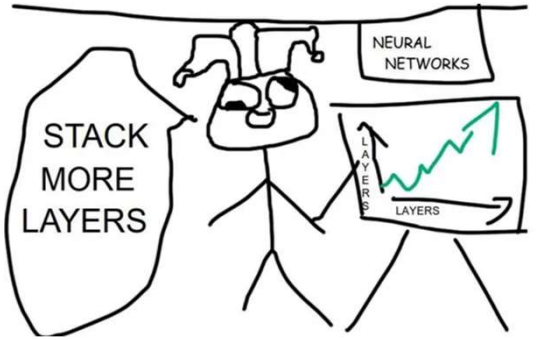
i

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ e & e & \exp(\alpha_3(z_1, z_2)) & 0 & 0 \\ e & e & e & \exp(\alpha_4(z_1, z_2)) & 0 \\ e & e & e & e & \exp(\alpha_5(z_1, z_2)) \end{pmatrix} \begin{matrix} x_1 = z_1 \\ x_2 = z_2 \\ b_3(z_1, z_2) + \exp(\alpha_3(z_1, z_2)) \cdot z_3 \\ b_4(z_1, z_2) + \exp(\alpha_4(z_1, z_2)) \cdot z_4 \\ b_5(z_1, z_2) + \exp(\alpha_5(z_1, z_2)) \cdot z_5 \end{matrix}$$

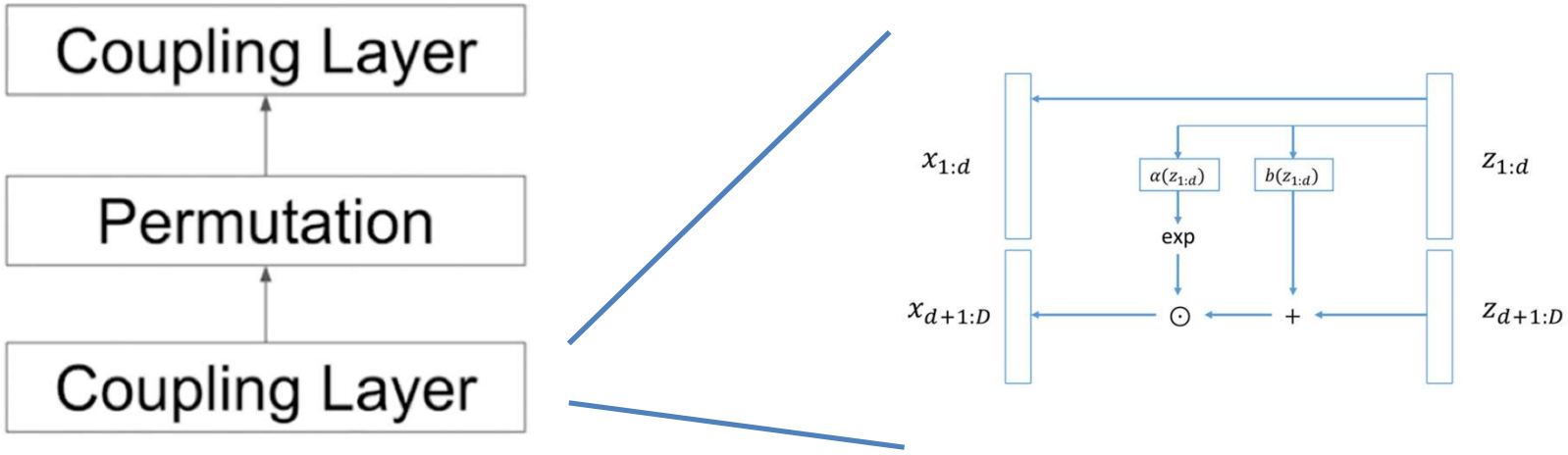
$$\frac{\partial .}{\partial z_1} \quad \frac{\partial .}{\partial z_2} \quad \frac{\partial .}{\partial z_3} \quad \frac{\partial .}{\partial z_4} \quad \frac{\partial .}{\partial z_5}$$

e=don't care

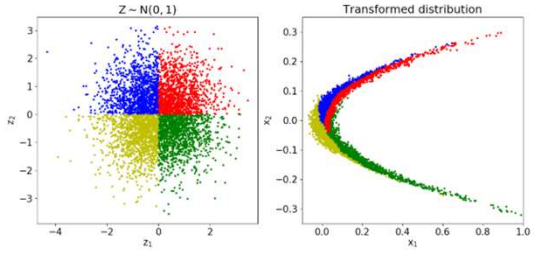
Stack more Layers (Permutation)



- In RealNVP
  - d is arbitrary and also the ordering
- When stacking several coupling layers put fixed permutation of dimensions in between
- Fix use a permutation
  - is invertible and det=1



Example



Listing 6.realnvp: The simple example in TFP

```
#12345678901234567890123456789012345678901234567890123456789012345678901234
```

```
bijectors=[] #A


h = 32
for i in range(5): #B
    net = tfb.real_nvp_default_template(hidden_layers=[h, h])#C
    bijectors.append(tfb.RealNVP(shift_and_log_scale_fn=net,num_masked=num_masked))#D
    bijectors.append(tfb.Permute([1,0])) #E
    self.nets.append(net)
bijector = tfb.Chain(list(reversed(bijectors[:-1])))


self.flow = tfd.TransformedDistribution(#F
    distribution=tfd.MultivariateNormalDiag(loc=[0., 0.]),
    bijector=bijector)
```

Glow for image data

--arXiv:1807.03039

Glow: Generative Flow with Invertible 1×1 Convolutions

Diederik P. Kingma*, Prafulla Dhariwal*
OpenAI, San Francisco

Specialties of glow

- Use 1x1convolutions instead of Permutation

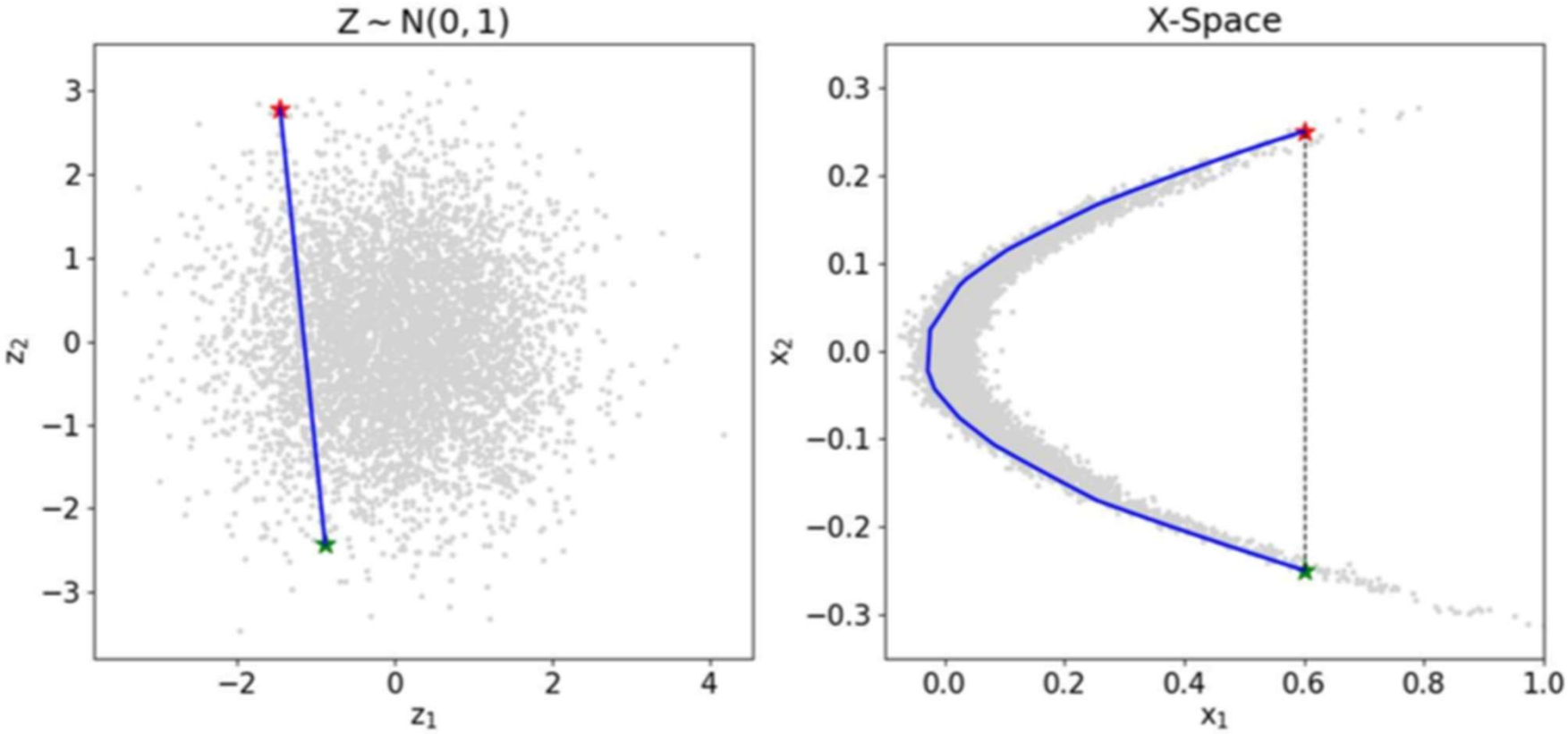
- Image Data
  - Multiscale Architecture (also in RealNVP Paper)
  - X and Z are now tensors (3 dimensional, shape w,h,c)
  - Keep the w,h dimension work on the channel dimension
  - The channel dimension get's larger by squeeze operation (see below)
  - As before (Affine coupling layer now with tensors)

Demo

- Network has been trained on CelebA-HQ
  - 30000 (256x256x3) images of celebrities
  - Images have been aligned
- Sampling: draw 256*256*3 numbers from N(0,1)
  - Reduced Temperature draw from N(0,T*1)
- Interpolation
  - Blackboard
- Demo
  - Uses pretrained network
  - **fun_with_glow**

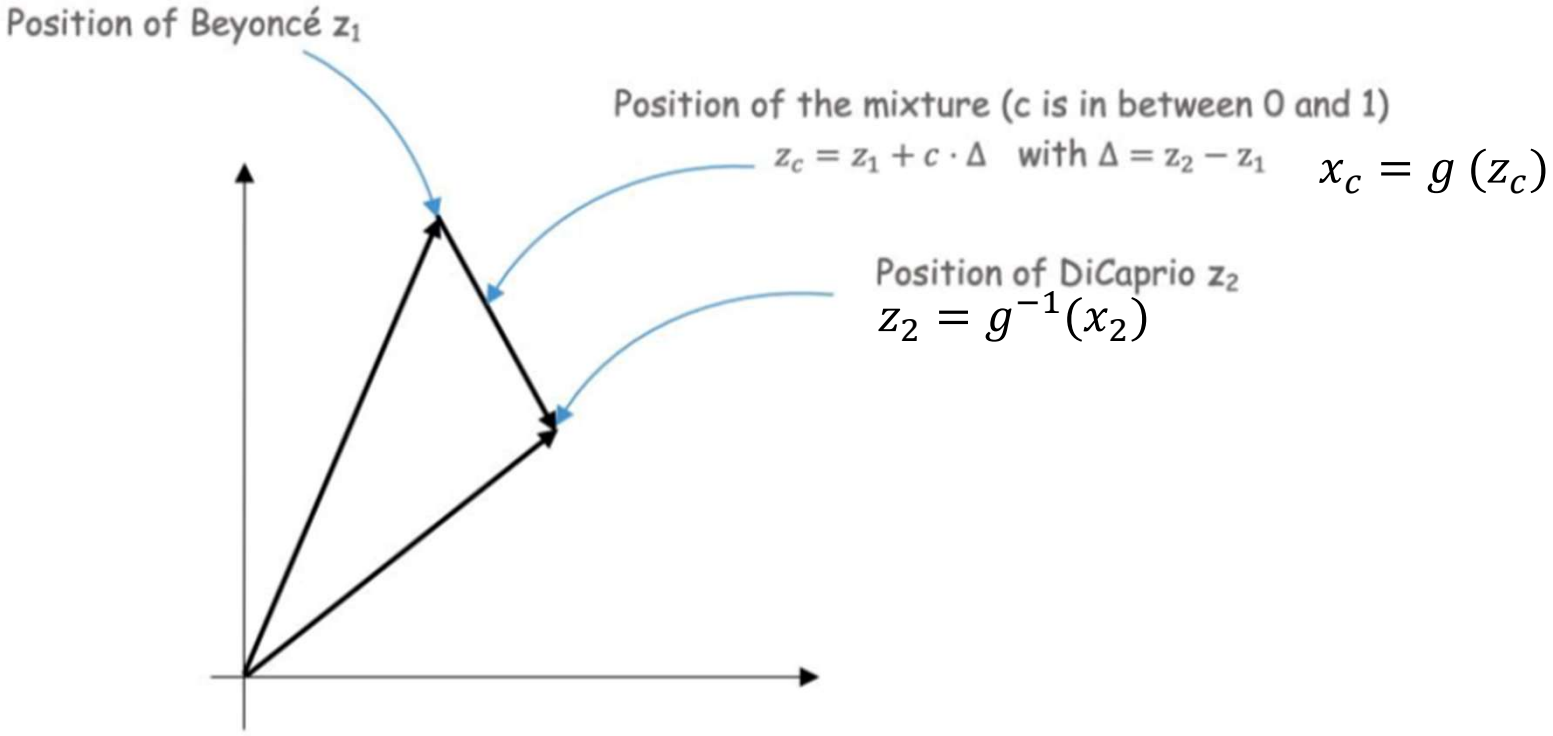
https://github.com/tensorchiefs/dl_book/blob/master/chapter_06/nb_ch06_05.ipynb

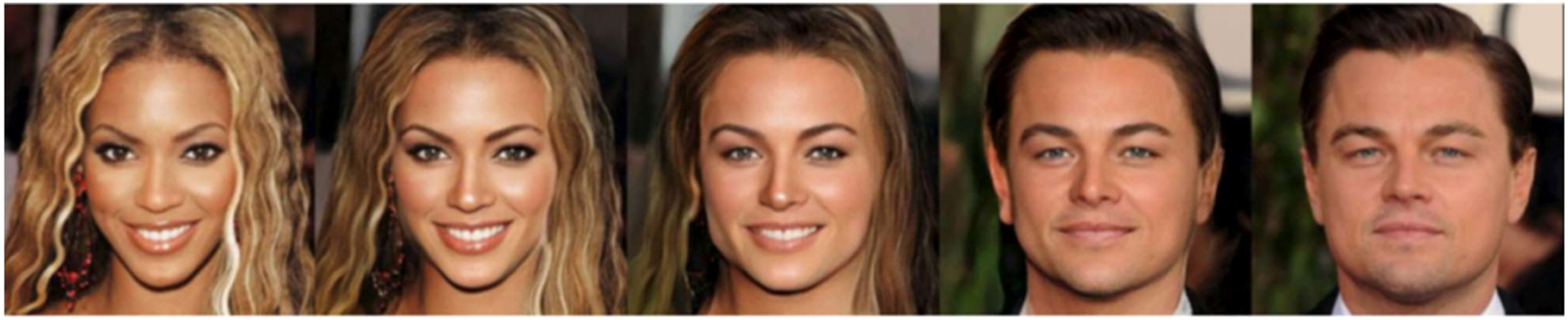
Morphing in 2-D



Morphing in 256x256x3 D

$z_1 = g^{-1}(x_1)$





Further reading

Some interesting reads and talks

- Eric Jang
  - Blog: part1 (introduction) part2 (modern flows)
  - 2019 ICML Tutorial
- Priyank Jaini
  - Lecture Waterloo University CS 480_680 8/24/2019 lecture 23 (slides | youtube)
  - SOS paper ICML (https://arxiv.org/abs/1905.02325) Talk
- Arsenii Ashukha
  - Lecture at day 3 at deepbayes.ru summer school 2019 (slides | video)
- Papers
  - Density estimation using Real NVP: https://arxiv.org/abs/1605.08803
  - Glow: Generative Flow with Invertible 1×1 Convolutions https://arxiv.org/abs/1807.03039